Mach Learn (2010) 80: 33–62
DOI 10.1007/s10994-010-5168-9

Finding a short and accurate decision rule in disjunctive normal form by exhaustive search

Peter R. Rijnbeek · Jan A. Kors

Received: 14 November 2007 / Revised: 16 December 2009 / Accepted: 28 December 2009 /
Published online: 29 January 2010

Abstract Greedy approaches suffer from a restricted search space which could lead to suboptimal classifiers in terms of performance and classifier size. This study discusses exhaustive search as an alternative to greedy search for learning short and accurate decision rules. The Exhaustive Procedure for LOgic-Rule Extraction (EXPLORE) algorithm is presented, to induce decision rules in disjunctive normal form (DNF) in a systematic and efficient manner. We propose a method based on subsumption to reduce the number of values considered for instantiation in the literals, by taking into account the relational operator without loss of performance. Furthermore, we describe a branch-and-bound approach that makes optimal use of user-defined performance constraints. To improve the generalizability we use a validation set to determine the optimal length of the DNF rule. The performance and size of the DNF rules induced by EXPLORE are compared to those of eight well-known rule learners. Our results show that an exhaustive approach to rule learning in DNF results in significantly smaller classifiers than those of the other rule learners, while securing comparable or even better performance. Clearly, exhaustive search is computer-intensive and may not always be feasible. Nevertheless, based on this study, we believe that exhaustive search should be considered an alternative for greedy search in many problems.

Keywords Rule learning · Induction · Exhaustive search · Branch-and-bound

1 Introduction

Decision rules are among the most expressive and comprehensible knowledge representation formalisms. Many different approaches to learn decision rules have been developed (Mitchell 1997). One approach is to first generate a decision tree from which a rule set is then derived (Quinlan 1992). The decision tree is induced by using a "divide-and-conquer"

Editor: Johannes Fürnkranz.

P.R. Rijnbeek (✉) · J.A. Kors
Department of Medical Informatics, Erasmus University Medical Center, P.O. Box 2040, 3000 CA Rotterdam, The Netherlands
e-mail: p.rijnbeek@erasmusmc.nl

strategy in which the feature space is recursively partitioned until regions with examples of mainly the same class are obtained. In a subsequent step rules are derived directly from each branch of the tree. Finally, a global optimization step is applied to prune the rule set to improve classification accuracy.

Another rule learning approach is to use a "separate-and-conquer" strategy in which an ordered set of rules is generated by first learning the single best conjunctive rule on the training set. After removal of the positive examples covered by this rule, the process is iterated over the remaining training examples. This approach, also called sequential covering, was first used in the AQ family of algorithms developed by Michalski et al. (1986) but has since then been used by many rule learners (Clark and Niblett 1989; Cohen 1995; Weiss et al. 1990; Frank and Witten 1998). This method can be used to generate two types of rule sets: ordered and unordered. In an ordered rule set, rules can only be interpreted in the context of previous rules, while in an unordered rule set each rule can be interpreted independently. RIPPER for example applies sequential covering combined with a reduced error pruning strategy (Cohen 1995) to induce ordered or unordered rule sets. A small number of cases from the training set is used to extract a rule based on the minimum description length principle. In two-class problems RIPPER only induces rules for one of the classes, which makes their order irrelevant. In PART the decision tree approach and sequential covering are combined by repeatedly generating partial decision trees (Frank and Witten 1998). In essence, a pruned decision tree is built for the set of examples remaining after the previous covering step, the leave with the largest coverage is made into a rule, and the rest of the tree is discarded.

A disadvantage of both "divide-and-conquer" and "separate-and-conquer" is that, as induction progresses, the number of available training examples dwindles. As a result decisions are being made with less and less statistical support. To overcome this problem, an approach called "conquering-without-separating" has been proposed by Domingos (1994). Essentially, this approach conducts a batch, hill-climbing, specific-to-general search through the space of rule sets. The algorithm is initialized by generating a specific rule for each case in the dataset and applies a heuristic evaluation function to prune the rule set. The final classification is achieved through weighted voting among the remaining rules.

Although all these approaches have been shown to perform well, they do have their limitations. Firstly, they are based on heuristics to reduce the search space, and thus may miss the classifier that performs best. Secondly, the resulting classifiers are often unnecessarily complex, that is, the same classification accuracy can be obtained with a smaller, more comprehensible classifier (Rückert and De Raedt 2008). Comprehensibility is an important aspect of a classifier, for example in medical applications in which users want to understand the ways by which a classification result is reached. Algorithms that generate ordered rule sets may suffer from poor comprehensibility because an individual rule cannot be interpreted without consideration of the previous rules in the covering sequence. A single rule in Disjunctive Normal Form (DNF) would be far easier to understand. Furthermore, less complex classifiers are preferable according to Occam's razor principle, which states that for the explanation of any phenomenon one should make only as many assumptions as needed. Unnecessary complexity can reduce the generalizability of the classifier. A third limitation of most existing rule learners is that they aim to maximize the accuracy of the classifier. However, in many application areas, for example in medicine, sensitivity and specificity are more pertinent measures for the characterization of classification performance (Galen and Gambino 1975). A classifier in these areas often needs to be maximal for either sensitivity or specificity while fulfilling a user-specified minimum performance for the other. For instance, in classifying a life-threatening disease in a hospital

environment the highest possible sensitivity at an acceptable, minimum specificity level may be required. Some rule learning algorithms allow for cost-sensitive learning with the help of a cost matrix that directs the sensitivity-specificity trade-off. Cost-insensitive algorithms can be made cost-sensitive by resampling or wrapper techniques (Elkan 2001; Viaene and Dedene 2005), such as MetaCost (Domingos 1999). However, the choice of the cost parameters is far from straightforward and generally requires a process of trial-and-error.

These limitations of heuristic search can be overcome by an exhaustive search of the feature space. Clearly, when all possible classifiers are scrutinized, it is easier to find the best classifier with least complexity, and optimization under constraints is straightforward. The problem, of course, is that the execution time of a truly exhaustive search algorithm may soon become prohibitive even for moderately sized problems. Furthermore, there is the problem of overfitting. This could be handled by applying a proper stop criterion on the search depth (Dietterich 1995). We previously developed an algorithm called EXPLORE (Exhaustive Procedure for LOgic-Rule Extraction), that exhaustively searched for the smallest DNF rule that fulfills user-specified performance requirements (Kors and Hoffmann 1997). In this algorithm the rules were generated exhaustively using a time-consuming, brute-force approach. This strongly limited its practical applicability. In the present study, we propose several new techniques that allow for a considerably more efficient exhaustive search. We show that a new version of EXPLORE which incorporates these techniques is comparable, in terms of accuracy, to a wide range of rule learners while being able to induce considerably smaller classifiers.

1.1 Related work

The simplest exhaustive algorithm is the 1R algorithm which searches exhaustively for the most informative single feature and bases a rule set on this single feature alone (Holte 1993). Such a very simple classifier was shown to perform surprisingly well on many datasets, but obviously more complex rules with potentially better performance are being missed.

A number of algorithms have been developed that approximate exhaustive search by massive beam-search, or that search exhaustively but for conjunctive rules only (Clark and Niblett 1989; Segal 1997; Weiss et al. 1990). In a beam search a fixed number, the beam width, of candidate rules are considered in each step of the covering approach. This enlarges the search space as compared to a hill-climbing approach, which has a beam width of one. Beam search is a massive-search heuristic if a large beam width is chosen, but it is not exhaustive because it does not generate all possible combinations of feature tests. A well-known general-to-specific beam search algorithm is CN2 (Clark and Niblett 1989) in which ideas from the AQ (Michalski et al. 1986) and ID3 (Quinlan 1986) algorithms are combined. CN2 applies the covering approach of AQ but relaxes the constraint that only rules be included that perform perfectly on the training data by applying pruning techniques as used in ID3. BRUTE is a massive beam search algorithm that induces conjunctive decision rules (Segal 1997). BRUTE was shown to perform better than greedy search algorithms on a large number of datasets. The algorithm allows the user to specify a given performance measure to be optimized, but it is not possible to simultaneously impose constraints on other performance measures. The OPUS system searches exhaustively for conjunctive rules that optimize the Laplace accuracy (Webb 1995). Using an admissible search strategy, OPUS reduces the search space to a manageable size without loss of performance.

All these conjunctive rule learners can be employed as the primary rule learner in a sequential covering approach. For example, BRUTE is implemented in BRUTEdl (Segal

and Etzioni 1994), which generates a rule set. However, as discussed above, ordered rule lists are less comprehensible than a single DNF rule. The Predictive Value Maximization (PVM) algorithm developed by Weiss et al. (1990) generates rules in DNF format based on a beam search strategy. Weiss demonstrated that a massive search strategy often outperforms a greedy search. An attractive feature of the PVM algorithm is that it allows optimization under constraints. The Stochastic Local Search algorithm SL^2 induces DNF rules by means of a stochastic approach that is claimed to provide a near-optimal solution (Rückert and De Raedt 2008). The SL^2 algorithm explicitly aims at inducing small rule sets with few literals, and it was shown that many rule learning systems produce unnecessarily large rule sets compared to SL^2 (Rückert and De Raedt 2008). Unfortunately, their SL^2 algorithm can only handle nominal or discrete attributes while many datasets contain continuous-valued attributes.

Finally, Classification Association Rule Mining (CARM) is an approach to classifier generation that builds a classifier by making use of Classification Association Rules (CARs), i.e., association rules whose right-hand-side is restricted to the class attribute (Yin and Han 2003; Liu et al. 1998). Bayardo (1997) use a brute-force approach to mine conjunctive decision rules based on an association rule learner, enhanced with new pruning techniques. Since heuristic pruning techniques are applied and the rules are induced given some support and confidence constraints, the search is only approximately exhaustive. The Classification Based on Association (CBA) algorithm is a massive search algorithm that generates all the frequent rule items (Liu et al. 1998). CBA applies a covering approach to build the final classifier and uses heuristics to reduce the search space. A general disadvantage of the association rule based algorithms is that continuous-valued attributes can only be handled after a discretization step.

The present paper is organized as follows: In Sect. 2 we describe the EXPLORE algorithm in detail. The scalability of the algorithm is considered in Sect. 3. In Sect. 4, EXPLORE is compared with the other rule learners with respect to performance and classifier size. Finally, in Sect. 5 we discuss our findings and present directions for future research.

2 Exhaustive procedure for logic-rule extraction

2.1 Preliminaries

We first define some basic concepts used throughout the article.[1]

Definition 1 (Example) A labeled example $x_k \in X$ consists of values for all $f \in F$ features and a class label $c \in \{0, 1\}$. An example x_k is said to be positive if $c_k = 1$, and negative if $c_k = 0$. X is divided in a training set, X_{train}, and a testing set, X_{test}.

Definition 2 (Literal) A literal l is a feature-operator-value triad, $l = (f, o, v)$, in which f is a feature, o is a relational operator $(\leq, =, >)$, and v is a value. An example x_k satisfies a literal $l = (f, o, v)$ if the value of feature f in x_k compared through the relational operator o with the value v evaluates to *True*.

Definition 3 (Term) A term is a conjunction of literals. The term size t denotes the number of literals in a term.

[1] In Appendix A, a table is given of the defined symbols.

Table 1 Estimation of performance measures from a 2×2 classification matrix

	Rule positive	Rule negative
Class positive	True positives (TP)	False negatives (FN)
Class negative	False positives (FP)	True negatives (TN)

Sensitivity $= TP/(TP + FN)$
Specificity $= TN/(FP + TN)$
Positive predictive value $= TP/(TP + FP)$
Negative predictive value $= TN/(FN + TN)$
Accuracy $= (TP + TN)/(TP + FN + FP + TN)$

Definition 4 (Rule) A formula in Disjunctive Normal Form (DNF) is a disjunction of terms. A rule r is a DNF formula that is associated with a class $c \in \{0, 1\}$, taking the form: *if* (*DNF formula*) *then* $c = 1$ *else* $c = 0$. The length n of a rule is the number of literals in a rule.

We wish to point out that in the literature on covering algorithms, a rule is often defined as consisting of a single term, and disjunctions of terms are represented as a rule set (Cohen 1995; Clark and Niblett 1989). In our definition a rule is a DNF formula which can contain more than one term.

Definition 5 (Performance measure) The performance measure P that needs to be optimized is one of the measures as defined in Table 1. By $P(r, X)$ we denote the performance of rule r on the example set X.

Definition 6 (Performance constraints) The performance constraints C are minimal constraints on one or more of the performance measures, other than P, that should be attained by a rule. If the performance constraints are fulfilled for rule r on example set X, then $C(r, X) = True$, else $C(r, X) = False$.

2.2 Induction process

To induce a DNF rule from examples we apply the following steps:

i. *Performance specification.* A performance measure P is selected, and minimum constraints C are specified for the remaining performance measures.
ii. *Initialization.* To determine the optimal rule length, an embedded hold-out approach is used. From X_{train} 2/3 of the examples are randomly selected as a learning set X_{learn} and the remaining 1/3 of the examples are taken as a validation set $X_{validate}$. The rule length n is initialized to 1.
iii. *Rule generation.* For X_{learn}, EXPLORE systematically generates all rules of length n. The best rule b_n is selected, i.e., the rule r for which $P(r, X_{learn})$ is maximal and $C(r, X_{learn}) = True$.
iv. *Stop criterion.* If $P(b_n, X_{validate}) > P(b_{n-1}, X_{validate})$ then n is incremented by one and step (iii) is repeated. If not, n is taken as the optimal rule length, n_{opt}.
v. *Evaluation.* For X_{train}, EXPLORE determines the best decision rule of length n_{opt}, i.e., the rule r for which $P(r, X_{train})$ is maximal and $C(r, X_{train}) = True$. The true performance of this rule is estimated on X_{test}.

We will now focus on step (iii), the exhaustive generation of rules of length n, and propose a number of approaches to reduce the search space without loss of performance.

2.3 Exhaustive rule generation

First, we define the terminology used in the rule-generation procedures of EXPLORE.

Definition 7 (Literal tuple) A literal tuple $L = (l_1, \ldots, l_n)$ is an ordered list of literals in a rule of length n. The literal at position j in term i is denoted by $l_{i,j}$.

Definition 8 (Term tuple) A term tuple $T = (t_1, \ldots, t_m)$ is an ordered list of term sizes of the m terms of the rule.

For example, for $r = (A > 2 \wedge B = green) \vee A \leq 1$, $T = (2, 1)$ and $L = (A > 2, B = green, A \leq 1)$. Note that a rule r is completely defined by the literal tuple and the term tuple, $r(L, T)$, since the literal tuple defines all the literals and the term tuple defines how the literals should be combined logically.

Definition 9 (Feature-operator list) The set of feature-operators FO consists of a lexicographically ordered list of all the $(f, >)$ and $(f, \leq)$ pairs for continuous features, with $>$ preceding $\leq$, and all the $(f, =)$ pairs for nominal features. We denote the feature-operator in term i at position j as $fo_{i,j}$.

Definition 10 (Value list) The value list $V(fo)$ is a list of values for a feature-operator fo. The value lists of all feature-operators are denoted by V.

The rule generation step in EXPLORE is presented in Algorithm 1. The algorithm consists of an initialization part in which the lexicographically ordered feature-operator list and the value lists are generated (lines 1–2).

The main part of EXPLORE consists of three nested do-while loops, which systematically generate admissible decision rules of length n. The outer loop generates a term tuple (lines 6–17), the second loop instantiates the literal tuple with feature-operator pairs (lines 8–16), and the inner loop instantiates the literal tuple with values (lines 10–14). If r fulfills the performance constraints and outperforms the current best rule, this rule becomes the new best rule (lines 11–13).

To improve readability, the literal tuple L and the term tuple T are passed by reference to the functions. If one of these global variables are changed by a function this is denoted in the results section in the header. The functions in the do-while loops return *True* on success and *False* otherwise.

In the following paragraphs we will give a detailed description of each of the functions in Algorithm 1.

2.3.1 Generation of feature-operators

The lexicographically ordered list of feature-operators is generated as shown by Algorithm 2. The feature-operator list is appended with a $(f, =)$ pair for nominal features and with $(f, >)$ and $(f, \leq)$ pairs for continuous-valued features.

Algorithm 1: EXPLORE

Input : X, examples; F, features; n, rule length; P, performance measure; C, constraints.
Output: b, best rule of length n.

```
   // initialize rule generation
1  FO = GenerateFeatureOperatorList(F);
2  V = GenerateValueLists(X, FO);
   // start rule generation
3  T = (n);                          // start with one term of size n
4  L = ∅;
5  b = ∅;
6  do
7      InitFeatureOperators(FO, L, T);
8      do
9          if InitValues(L, T, V) then
10             do
                   // evaluate r(L, T)
11                 if (P(r, X) > P(b, X)) ∧ C(r, X) then
12                     b = r;          // best rule becomes current rule
13                 end
14             while NextValues(L, T, V)
15         end
16     while NextFeatureOperators(FO, L, T)
17 while NextTermTuple(T)
18 return b;
```

Algorithm 2: GenerateFeatureOperatorList

Input : F, features.
Output: FO, lexicographically ordered list of feature-operator pairs.

```
1  FO = ∅;
2  foreach f ∈ F do
3      if f is nominal then
4          FO ← (f, =);
5      else
6          FO ← (f, >);
7          FO ← (f, ≤);
8      end
9  end
10 return FO;
```

2.3.2 *Generation of list of values*

Algorithm 3 generates for each feature-operator a list of values that are used to instantiate the literals. The values are sorted in decreasing order by the number of true positives generated

Algorithm 3: GenerateValueLists

Input : X, examples; FO, feature-operator list.
Output: V, value lists for all feature-operator pairs.

```
foreach fo ∈ FO do
    V(fo) = ∅;
    if f in fo is nominal then
        V(fo) equals all nominal values;
    else
        V(fo) equals all values determined by subsumption;
    end
    sort V(fo) in decreasing order by number of true positives;
end
return V;
```

Table 2 Example dataset with one feature A

Example	A	Class
1	4.50	0
2	5.00	0
3	5.10	1
4	5.20	1
5	5.40	0
6	6.00	1

by a single literal containing the feature-operator and the value. For a nominal feature, the value list consists of all its nominal values. For a continuous-valued feature, a large reduction in values is possible if the principle of subsumption pruning, often used in beam-search strategies, is applied. For example, suppose we have the following two rules:

1. if $A > 5$ then $c = 1$ else $c = 0$
2. if $B > 7$ then $c = 1$ else $c = 0$

If rule 1 covers a superset of the positive examples covered by rule 2, and a subset of the negative examples, then rule 1 has a higher coverage of correctly classified examples. Any conjunctive extension of rule 1 with a second literal will perform at least as good as the extension of rule 2. In subsumption pruning a rule is removed from the set of promising rules if there exists another rule that covers a superset of the positive examples and a subset of the negative examples (Webb 1995).

The ideas behind subsumption pruning can also be applied to reduce the number of values without loss of performance. Suppose we have a small dataset ordered by increasing feature values of a feature A as shown in Table 2, and we want to determine only the relevant values.

A simplistic way to generate the values would be to take all averages of two subsequent feature values in the ordered value list for both feature-operators ($A >$ and $A \leq$). However, it is unnecessary to select values of examples that have adjacent feature values and assign to the same class, because class separation will never improve at those values. This class-boundary approach has been used earlier for greedy approaches (Fayyad and Irani 1992). However, a further reduction is possible if we apply the subsumption principle. To illustrate

Table 3 Number of true positives (*TP*) and false positives (*FP*) for literals that contain $A >$ and a value that is equal to the average feature value of adjacent cases

Literal	*TP*	*FP*
$A > 4.75$	3	2
$A > 5.05$	3	1
$A > 5.15$	2	1
$A > 5.30$	1	1
$A > 5.70$	1	0

Table 4 Number of values for standard datasets using the class-boundary approach and the subsumption principle. The percentage reduction in the number of values is indicated in parentheses

Dataset	Examples	Features		Values		
		Continuous	Nominal	Boundary	Subsumption	
australian	690	6	8	1648	1128	(31.6%)
breast-cancer	277	0	10	41	41	(0.0%)
breast-w	683	9	0	152	138	(9.2%)
diabetes	768	8	0	1736	1220	(29.7%)
glass(G2)	214	9	0	654	380	(41.9%)
heart-statlog	270	7	6	578	426	(26.3%)
liver	345	7	0	556	448	(19.4%)
mammographic	830	3	2	143	127	(11.2%)
mushroom	5644	0	22	98	98	(0.0%)
sick	2643	6	21	1180	954	(19.2%)
vote	232	0	16	32	32	(0.0%)
wine	178	13	0	1202	732	(39.1)

this we calculate the number of true positives (*TP*) and false positives (*FP*) for all literals that contain the feature-operator "$A >$" and a value that is equal to the average feature value of adjacent examples, as shown in Table 3.

Based on the subsumption principle, only those literals have to be considered that cover a superset of the positive examples and a subset of the negative examples. Thus, in our example, $A > 5.05$ is preferred over $A > 4.75$, because there are less *FP* for the same number of *TP*; $A > 5.05$ is preferred over $A > 5.15$ and $A > 5.30$, because there are more *TP* for the same number of *FP*. In our example the final relevant values for feature-operator $A >$ are 5.05 and 5.70. Generalizing, for the relational operator $>$, only values that are at a class boundary from 0 to 1 have to be taken as values. Similarly, for the $\leq$ operator we only need to include values that are at a class boundary from 1 to 0. However, if there are examples with the same feature value but different labels, adjacent values are relevant for both relational operators. Note that if we would not apply the subsumption principle, the class-boundary approach would result in 3 values for each feature-operator.

Table 4 shows the effect of the subsumption selection strategy on the total number of values of all feature-operators for 12 standard datasets taken from the UCI data repository (Murphy and Aha 1994). Examples with missing values have been removed.

The percentage reduction in number of values strongly varies across datasets. Since the subsumption strategy applies to continuous-valued features, datasets that only contain nominal features obviously have no reduction in values. The datasets that showed the largest

Table 5 Enumeration of all term tuples for rule length $n = 5$

(5)
(4, 1)
(3, 2)
(3, 1, 1)
(2, 2, 1)
(2, 1, 1, 1)
(1, 1, 1, 1, 1)

reduction were glass (41.9%) and wine (39.1%); the smallest reduction was seen for mammographic (11.2%) and breast-w (9.2%).

2.4 Generation of term tuples

The generation of term tuples can be seen as the problem of generating all finite decreasing sequences of positive integers $(t_1, t_2, \ldots, t_m)$ such that $\sum_{i=1}^{m} t_i = n$. This is called the partitioning of a positive integer n (Andrews and Eriksson 2004). As an example, Table 5 shows all term tuples for $n = 5$.

2.4.1 Generation of next term tuple

The first term tuple is initialized to a single term of size n, which equals the rule length (Algorithm 1). The next term tuple is systematically generated by Algorithm 4. In integer partitioning theory, several algorithms exist that generate a new partitioning from a given partitioning $t_1, t_2, \ldots, t_m$ (Zoghbi and Stojmenovic 1998). Define h as the number of terms greater than 1, i.e., $t_i > 1$ for $1 \leq i \leq h$, and $t_i = 1$ for $h < i \leq m$. We use an algorithm that is based on the idea of subtracting one from the term $t_h > 1$ and replacing the terms to the right with as many terms c of size $(t_h - 1)$ as possible and a single term with a size of the remainder d (Algorithm 4). The last term tuple then consists of n terms of size 1, i.e., $t_1 = 1$.

2.5 Generation of feature-operators

After a term tuple has been generated, the literal tuple will be instantiated with features and operators (Algorithm 1). Several characteristics of decision rules can help to greatly reduce the number of those instantiations, without loss of performance.

Firstly, each feature-operator occurs only once per term, since multiple occurrences of the same feature-operator in a term are redundant, e.g., $A > AND\ A >$. Notice that the same feature can occur twice in a term with different operators, e.g., $A > AND\ A \leq$.

Secondly, permutation of feature-operators in a term is not necessary. For example, $A > AND\ B \leq$ is logically equivalent to $B \leq AND\ A >$. Therefore, we only instantiate a term with feature-operators in lexicographic order.

Thirdly, permutation of terms in a rule is not necessary. For example, $(A > AND\ B >)\ OR\ (C > AND\ D >)$ is logically equivalent to $(C > AND\ D >)\ OR\ (A > AND\ B >)$.

Fourthly, for continuous-valued features each feature-operator is used only once to instantiate a term of size 1, since $A > OR\ A >$ is always logically equivalent to single $A >$. However, $(A > AND\ B >)\ OR\ (A > AND\ B >)$ has to be generated because it cannot be simplified if the values in the literals are different.

Next we will describe the function InitFeatureOperators which initializes the literal tuple with feature-operators and the function NextFeatureOperators which generates the next instantiation of the literal tuple with feature-operators.

Algorithm 4: NextTermTuple

Input : T, term tuple.
Output: *True* if the next term tuple T could be generated, *False* otherwise.
Result : T equals the next term tuple.

```
1  m = |T|;                    // m equals the current number of terms
2  if t_1 > 1 then
       // check the size of the last term m
3      if t_m = 1 then
4          Replace t_h > 1, t_{h+1} = 1, ..., t_m = 1 by c integers equal to (t_h − 1), and a
           single integer d, such that 0 < d ≤ t_h − 1 and c(t_h − 1) + d = t_h + m − h;
5          m = c + h;
6      else
7          Replace t_1, ..., t_m by t_1, ..., t_m − 1, 1;
8          m = m + 1;
9      end
10     T = (t_1, ..., t_m);
11     return True;
12 else
13     return False;
14 end
```

Algorithm 5: InitFeatureOperators

Input : FO, feature-operator list; L, literal tuple; T, term tuple.
Result: L is instantiated with feature-operators.

```
   // Initialize feature-operators in all terms
1  for i = 1 to |T| do
2      InitFeatureOperatorsTerm(i, FO, L, T);
3  end
```

2.5.1 Initialization with feature-operators

The feature-operators in the literal tuple are initialized by the function InitFeatureOperators (Algorithm 5). For each term, it calls InitFeatureOperatorsTerm (Algorithm 6). If the size of a term is not equal to the size of the preceding term (line 1), the term is initialized to lexicographically ordered feature-operators (lines 2–4). If the term sizes are equal and greater than 1, the term is initialized by copying the instantiation of the preceding term (lines 7–9). If term sizes are equal to 1, the instantiation of the preceding term is copied for a nominal feature; for a continuous-valued feature, the term is instantiated with the lexicographically next feature-operator (lines 11–15).

2.5.2 Instantiation with next set of feature-operators

The function NextFeatureOperators instantiates a rule with the next set of feature-operators (Algorithm 7). Starting at the last term and moving to the left, we find the first term for

Algorithm 6: InitFeatureOperatorsTerm

Input : i, term number; FO, feature-operator list; L, literal tuple; T, term tuple.
Result: L is instantiated with feature-operators for term i.

```
1  if t_i ≠ t_{i-1} ∨ i = 1 then
       // instantiate feature-operators in lexicographic order
2      for j = 1 to t_i do
3          fo_{i,j} = FO_j;
4      end
5  else
6      if t_i > 1 then
7          for j = 1 to t_i do
8              fo_{i,j} = fo_{i-1,j};                    // copy previous term
9          end
10     else
11         if f in fo_{i,j} is nominal then
12             fo_{i,1} = fo_{i-1,1};                    // copy previous term
13         else
14             fo_{i,1} = fo_{i-1,1} + 1;          // take next feature-operator
15         end
16     end
17 end
```

Algorithm 7: NextFeatureOperators

Input : FO, feature-operator list; L, literal tuple; T, term tuple.
Output: *True* if L could be instantiated with the next set of feature-operators, *False* otherwise.
Result : L is instantiated with the next set of feature-operators.

```
1  for i = |T| to 1 do
2      if NextFeatureOperatorsTerm(i, FO, L, T) then
3          for k = i + 1 to |T| do
4              InitFeatureOperatorsTerm(k, FO, L, T);
5          end
6          return True;
7      end
8  end
9  return False;
```

which a next set of feature-operators can be generated. All the terms to the right of this term are initialized (lines 3–5).

Function NextFeatureOperatorsTerm (Algorithm 8) instantiates literal tuple L with the next set of feature-operators for term i. Starting at the last literal in the term and moving to the left, we find the first literal which is not instantiated with its last possible feature-operator. This literal is instantiated with the next feature-operator and all feature-operators to the right are initialized (lines 3–10).

Algorithm 8: NextFeatureOperatorsTerm

Input : i, term number; FO, feature-operator list; L, literal tuple; T, term tuple.
Output: *True* if L could be instantiated with the next set of feature-operators for term i, *False* otherwise.
Result : L is instantiated with the next set of feature-operators for term i.

```
// find literal that can be instantiated with a next
   feature-operator
```

for $j = t_i$ *to* 1 **do**
  $last = |FO| - t_i + j$;
  `// index of last possible feature-operator for literal` j
  if $fo_{i,j} \neq FO_{last}$ **then**
    $fo_{i,j} = fo_{i,j} + 1$;
    while $j < t_i$ **do**
      $j = j + 1$;
      $fo_{i,j} = fo_{i,j-1} + 1$;
    end
    return *True*;
  end
end
return *False*;

2.6 Instantiation of values

The next task is to instantiate the values in the literals which up to now consist only of feature-operators. Because the number of values is often very large compared to the number of feature-operators, especially for continuous-valued features, the total number of rules that are generated is mainly determined by the number of values per feature. We showed earlier that the number of values can be reduced considerably when subsumption is applied. To further reduce the search space at the value instantiation level we propose a branch-and-bound approach.

2.6.1 Value instantiation by branch-and-bound

Branch-and-bound searches the complete space of solutions iteratively by applying branching and bounding rules. The first operation of an iteration is branching, i.e., the unexplored solution space is subdivided into two or more subspaces to be investigated in a next iteration. Subsequently, a bounding rule for each of the subspaces is evaluated and compared to the current best solution. If it can be established that a subspace cannot contain the optimal solution, the subspace is discarded, else it is further explored. The search terminates when there is no unexplored part of the search space left. Branch-and-bound can provide a tightly focused traversal of the search space, while assuring that the optimal solution will be found (Webb 1995).

We systematically instantiate the literals in the rule only with values that have a potential to obtain a higher performance than the current best performing rule. To apply a branch-and-bound technique we need to define bounding rules. Clearly, the bounds are dependent on the performance measure to be optimized and on the user-defined performance constraints. We present here the bounding rules for two optimization problems: sensitivity optimization with a constraint on specificity, and accuracy optimization.

2.6.2 Sensitivity optimization with a constraint on specificity

In sensitivity optimization the number of true positives needs to be maximized. Let $TPmax_i$ be the maximum number of examples that can correctly be assigned to the positive class by term i, and let $TP_{i,j}$ be the number of true positives of literal j in term i. Then $TPmax_i$ is determined by:

$$TPmax_i = min(TP_{i,1}, \ldots, TP_{i,t_i}). \tag{1}$$

Note that the number of true positives of a term i, TP_i, can be lower than $TPmax_i$, because an example can be classified correctly by one literal and incorrectly by another. However, we can define a minimum bound $TPbound_i$ on $TP_{i,j}$, to guarantee that no rules will be generated that cannot improve on the number of true positives of the current best rule, *TPbest*. Suppose we have a rule that consists of one term $A > \wedge B >$. For a rule with one term the $TPbound_i$ is equal to *TPbest*. If literal $A > 5$ correctly classifies 10 positive cases ($TP_{1,1} = 10$) and $TPbest = 15$, then no instantiation of values in $B >$ can result in a decision rule with a higher number of true positives than *TPbest*. All conjunctive extensions of $A > 5$ can be skipped without loss of performance.

If we have a rule that consists of more than one term, then $TPbound_i$ depends on *TPbest* and the number of true positives generated by the other terms. Let $TPcum_i$ be the cumulative number of true positives of the rule up to term i. For example, if we have a term tuple $T = (2, 2, 2)$, $TPcum_2$ is the number of true positives of the disjunction of the first two terms. In EXPLORE we instantiate new values starting at the last literal in the last term. Therefore, each $TPcum_i$ can be calculated once and used as long as no literals are changed in that particular ensemble of terms. $TPbound_i$ can be calculated by taking into account the cumulative number of true positives up to term $i - 1$, $TPcum_{i-1}$, as well as the $TPmax_i$ of the terms $i + 1, \ldots, m$:

$$TPbound_i = TPbest - TPcum_{i-1} - \sum_{k=i+1}^{k=m} TPmax_k. \tag{2}$$

To reduce the search space even further, a second bound can be defined based on the specificity constraint. Let *FPbound* be the maximum number of false positives that are allowed according to the specificity constraint. The number of false positives of each term, FP_i, should be lower than *FPbound*, since an example can be classified as false positive by multiple terms. Note that *FPbound* is a constant value, in contrast to $TPbound_i$, which is updated after each generated value tuple.

We will now describe the initialization of the first completely instantiated rule and the subsequent generation of all relevant rules by branch-and-bound.

2.6.3 Initialization with values

The literal tuple is initialized by the function InitValues (Algorithm 9), which calls InitValuesTerm (Algorithm 10) for each term. InitValuesTerm initializes the literal tuple with values for term i. The values of each feature-operator were sorted in decreasing order by the number of true positives of a literal containing the feature-operator and the value (Algorithm 3). The literal with the highest number of true positives has the highest potential to improve on *TPbest*, therefore we take the first value in the value list as initial value for all literals (lines 1–3). If $TPbound_i$ is met we check *FPbound*, otherwise we return *False*

Algorithm 9: InitValues

Input : L, literal tuple; T, term tuple; V, set of values lists.
Output: *True* if L could be initialized with values, *False* otherwise.
Result : L is initialized with values.

```
1  i = 1;
2  while i ≤ |T| do
3      if InitValuesTerm(i, L, T, V) then
4          i = i + 1;
5      else
6          return False;
7      end
8  end
9  return True;
```

Algorithm 10: InitValuesTerm

Input : i, term index; L, literal tuple; T, term tuple; V, set of value lists.
Output: *True* if the literals in term i could be initialized with values, *False* otherwise.
Result : L is initialized with values in term i.

```
1  for 1 ≤ j ≤ t_i do
2      v_{i,j} = V(fo_{i,j})_1;
       // initialize to the first value in the list for fo_{i,j}
3  end
   // Branch and Bound
   // TPbound_i : minimum number of TPs for term i
   // FPbound : maximum number of FPs for each term
4  if TP_{i,j} ≥ TPbound_i then
5      if FP_i ≤ FPbound then
6          return True;
7      else
8          if NextValuesTerm(i, L, T, V) then
9              return True;
10         else
11             return False;
12         end
13     end
14 else
15     return False;
16 end
```

because a higher number of true positives will not be possible, i.e., for all feature-operators the values have been chosen with the highest number of true positives and $TPmax_i$ is determined by the literal with the lowest number of true positives (Eq. 1). If not $FP_i \leq FPbound$,

Algorithm 11: NextValues

Input : L, literal tuple; T, term tuple; V, set of value lists.
Output: *True* if L could be instantiated with a next set of values, *False* otherwise.
Result : L is instantiated with the next set of values.

```
1  i = |T|;
2  result = False;
3  while (i > 1) ∧ (result = False) do
4      if NextValuesTerm(i, L, T, V) then
5          result = True;
           // initialize terms to the right
6          k = i + 1;
7          while (k ≤ |T|) ∧ (result = True) do
8              result = InitValuesTerm(k, L, T, V);
9              k = k + 1;
10         end
11     else
12         i = i − 1;
13     end
14 end
15 return result;
```

we call NextValuesTerm (Algorithm 12), which generates the next set of values for this term.

2.6.4 Instantiation with next set of values

Function NextValues (Algorithm 11) instantiates the literal tuple with the next set of values that fulfill the bounds. It starts with the last term and calls NextValuesTerm (Algorithm 12) to instantiate the literal tuple with the next set of values for that term (line 4). If this is not possible we move to the left until a term is found that can be instantiated with a next set of values (line 12). If a term i can be instantiated we have to initialize all terms to its right again (lines 6–10). However, it is possible that one of these terms cannot be initialized to a value that fulfills the bounds. In that case, we determine the next values for term i and try again. The function returns *True* if a rule is found that fulfills both bounds and *False* otherwise.

Function NextValuesTerm (Algorithm 12) instantiates the literal tuple with the next set of values for a term. We start with the last literal (line 1) and move to the left, and reset the value to the first in the list, (line 2–5) as long as a literal contains the last value in the value list ($|V(fo_{i,j})|$) or $TP_{i,j} < TPbound_i$. We then instantiate the next value for the literal where we stopped (line 7). If the bounds are not met for the instantiated term we call NextValuesTerm again (line 8–12).

2.6.5 Accuracy optimization

The branch-and-bound approach can also be applied for the optimization of the accuracy of a DNF rule. Accuracy is defined as the total number of true classifications, $TT = TP + TN$, divided by the total number of examples N. We denote *TTbest* as the total number of correctly classified examples by the current best rule. Suppose that all negative examples (N_{neg})

Algorithm 12: NextValuesTerm

Input : i, item index; L, literal tuple; T, term tuple; V, value list.
Output: *True* if L could be instantiated with a next set of values for term i, *False* otherwise.
Result : L is instantiated with a next set of values for term i.

```
1  j = t_i;                                   // start with last literal
   // find literal that is not at its last value and for
       which TP_{i,j} ≥ TPbound_i
2  while j ≥ 1 ∧ (v_{i,j} = V(fo_{i,j})_{|V(fo_{i,j})|} ∨ TP_{i,j} < TPbound_i) do
3      v_{i,j} = V(fo_{i,j})_1;                        // reset to first value
4      j = j − 1;                          // move one literal to the left
5  end
6  if j ≥ 1 then
7      v_{i,j} = V(fo_{i,j})_{++};                     // instantiate next value
8      if TP_i ≥ TPbound_i ∨ FP_i ≤ FPbound then
9          return True;
10     else
11         return NextValuesTerm(i, L, T, V);
12     end
13 else
14     return False;
15 end
```

would have been correctly classified then we need at least $(TTbest - N_{neg})$ true positives to improve on the current best rule. $TPbound_i$ for each term can thus be calculated by taking into account the cumulative number of true positives up to term $i-1$, $TPcum_{i-1}$, as well as the $TPmax_i$ of the terms on the right of term i:

$$TPbound_i = (TTbest - N_{neg}) - TPcum_{i-1} - \sum_{i+1}^{i=m} TPmax_i. \tag{3}$$

Using the same approach *FPbound* can be derived for accuracy optimization. If we assume that all positive examples (N_{pos}) are correctly classified we need at least $(TTbest - N_{pos})$ true negatives which implies that:

$$FPbound = N_{neg} - (TTbest - N_{pos}) = N - TTbest. \tag{4}$$

The same algorithm can now be employed as described for sensitivity optimization with a specificity constraint. The bounds for accuracy optimization are less strong than for sensitivity optimization with a constraint on specificity because they are based on the extreme case that all negative or positive examples are classified correctly.

3 Scalability

In the previous paragraphs we have proposed several methods to strongly reduce the search space without introducing any performance degradation. This allows us to induce larger

rules in the same time. Nevertheless, exhaustive search will still be a computer-intensive approach. Clearly, the rule length is an important factor in the complexity of the problem. Furthermore, the complexity strongly depends on the dataset composition, i.e., the number of features and values will have a large impact on the size of the search space. In this section we will give some insight in the complexity by discussing the number of term tuples and literal tuples as a function of the rule length. Subsequently we will present the execution time of EXPLORE for experiments with the datasets used before.

3.1 Term tuples

The number of term tuples equals the number of logical combinations of terms in a rule of length n. Let $p(n,k)$ be the number of term tuples with all term sizes $t_i \geq k$. The total number of term tuples, $p(n,1)$, can then be calculated by the following recursive formula (Andrews and Eriksson 2004):

$$\begin{aligned} p(n,k) &= p(n,k+1) + p(n-k,k) \quad \text{if } (k < n), \\ p(n,k) &= 1 \qquad\qquad\qquad\qquad\quad \text{if } (k = n), \\ p(n,k) &= 0 \qquad\qquad\qquad\qquad\quad \text{if } (k > n). \end{aligned} \tag{5}$$

Figure 1 shows the exponential behavior of the number of term tuples as a function of the rule length.

3.2 Literal tuples

The number of literal tuples as function of the rule length depends on the number of feature-operator instantiations and the number of value instantiations in the literal tuple. Note that

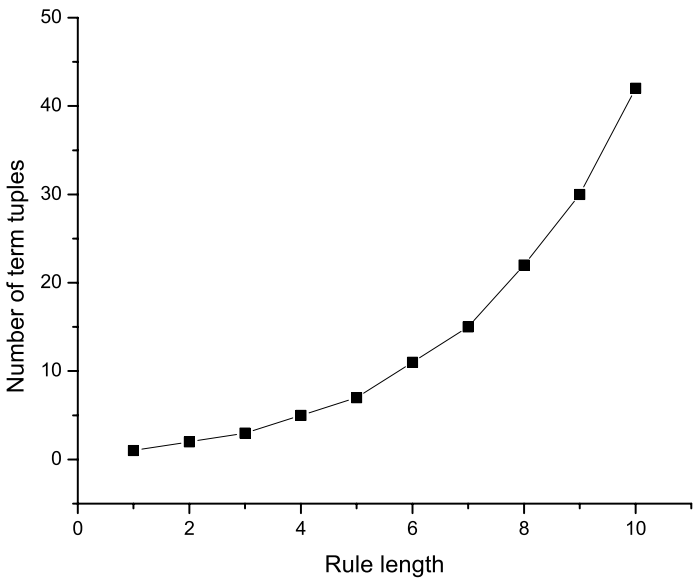

Fig. 1 Number of term tuples as a function of rule length

the total number of literal tuples equals the number of fully instantiated rules. Unfortunately, it is not possible to calculate the exact number of literal tuples as function of the rule length because the effectiveness of the branch-and-bound approach for value instantiation is strongly dependent on the data and user-defined performance constraints. However, a worst-case estimate can be derived.

Each feature-operator can occur only once in a single term. For a term of size t, the number of possible feature-operator instantiations with a set of $|FO|$ different feature-operators is therefore equal to $\binom{|FO|}{t}$. Since all terms have different sizes the total number of feature-operator instantiations is the product of the number of feature-operator instantiations for the individual terms. However, a rule can have terms of the same size. EXPLORE generates terms of equal size $t > 1$ with replacement and without ordering from the set of $\binom{|FO|}{t}$ possible feature-operator combinations (cf. Algorithm 7). The number of feature-operator instantiations $p_{t>1}$ for terms of equal size k, can be calculated by:

$$p_{t>1}(k, |FO|, t) = \binom{k + \binom{|FO|}{t} - 1}{k}. \tag{6}$$

Finally, in multiple terms of size $t = 1$, each feature-operator containing a continuous-valued feature is allowed only once. However, multiple occurrences of the same nominal features are possible. If the feature set FO consists of F_{nom} nominal features and F_{cont} continuous-valued features, then the number of feature-operator instantiations for k terms of size 1 is equal to

$$p_{t=1}(k, |F_{nom}|, |F_{cont}|) = \binom{|F_{cont}|}{k} + |F_{nom}|^k. \tag{7}$$

If a rule consists of terms of different sizes, the total number of instantiations is obtained by multiplying the number of feature-operator instantiations for terms of equal sizes, e.g., for 4 continuous-valued feature-operators and a term tuple $T = (3, 2, 2)$ the number of feature-operator tuples is $p_{t>1}(1, 4, 3) \times p_{t>1}(2, 4, 2) = 84$.

Up to now we only calculated the number of feature-operator instantiations for a specific term tuple. The total number of feature-operator instantiations for a certain rule length can be calculated by repeating the calculation described above for each term tuple. Figure 2 shows the number of instantiations in the literal tuple as a function of rule length, for different numbers of continuous-valued feature-operators.

Finally, to calculate the total number of literal tuples the number of value instantiations should be known. However, the branch-and-bound approach does not allow to compute this number analytically. We can only define an upper bound on the number of value instantiations. In a worst-case scenario the literals have to be instantiated by all values of each feature-operator pair, i.e., the branch-and-bound approach has no effect. The total number of value instantiations in each DNF rule can then be calculated by multiplying the number of values of all feature-operator pairs. If we assume that all the feature-operator pairs have an equal number of values c, then the number of feature-operator instantiations in a rule of length n as shown in Fig. 2 should be multiplied by c^n to get the total number of literal tuples.

3.3 Execution time

In Table 6 we present the execution times of EXPLORE for increasing rule length when optimizing on accuracy in a resubstitution experiment. We increased the rule length until

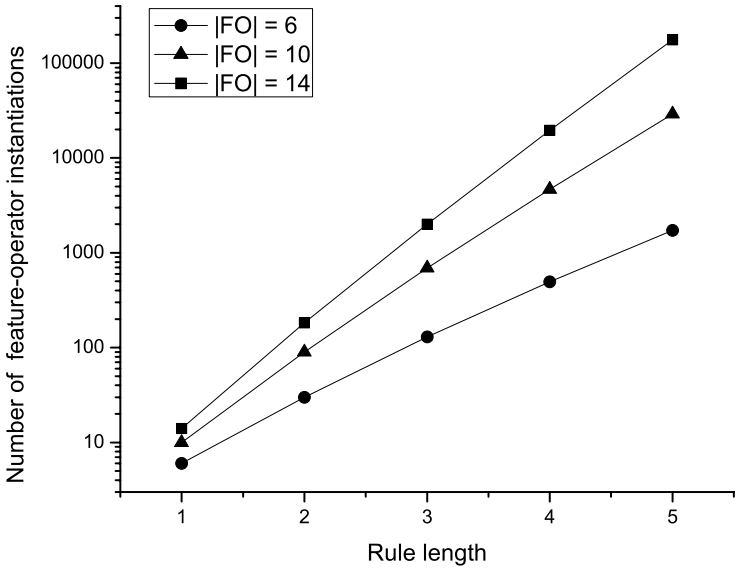

Fig. 2 Number of feature-operator instantiations as function of rule length, for 6, 10, and 14 continuous-valued feature-operators

the execution time became longer than 24 hours. The experiments were done on an Intel Xeon CPU 2.33 GHz with 3 GB of RAM.

As expected, execution time increases very fast with rule length. Moreover, the execution time strongly depends on the dataset composition. In datasets with predominantly nominal features, rules are induced faster than in datasets with many continuous-valued features. The number of examples is of less importance for nominal datasets, as shown by the large mushroom dataset. The main bottleneck appears to be the total number of values to instantiate the literals.

The EXPLORE algorithm is generally faster when optimizing sensitivity with a minimum constraint on specificity because much tighter bounds can be applied in the branch-and-bound approach. For example, the execution time of EXPLORE for the heart-statlog dataset for rules of length 4 is approximately 3.5 minutes when optimizing sensitivity with a specificity bound of 95% compared to 1.3 hours when optimizing accuracy. However, the reduction in execution time strongly depends on the dataset, the rule length, and the specificity constraint imposed.

4 Experimental evaluation

We compared the accuracy and the size of the DNF rule induced by EXPLORE with those of the classifiers induced by eight well-known rule learners. These eight algorithms represent all different rule learning strategies that have been described in the introduction to this paper:

- C4.5R (Quinlan 1992): induces a set of rules with a single term by applying a post-processing step on the decision tree learned by the C4.5 learning algorithm.
- CBA (Liu et al. 1998): is a CARM algorithm that induces classification rules based on an association rule learner. The implementation of Coenen was employed (Coenen 2004),

Table 6 Execution time (hour:min:sec) of a resubstitution experiment of different datasets for increasing rule length when optimizing accuracy

Dataset	Examples	Features		Values	Rule length						
		Continuous	Nominal		1	2	3	4	5	6	7
australian	690	6	8	1128	0:00:00	0:00:01	0:01:23	10:32:25			
breast-cancer	277	0	10	41	0:00:00	0:00:00	0:00:01	0:00:05	0:01:21	0:24:16	6:33:36
breast-w	683	9	0	152	0:00:00	0:00:00	0:00:05	0:02:09	1:35:36		
diabetes	768	8	0	1220	0:00:00	0:00:03	0:30:23				
glass(G2)	214	9	0	380	0:00:00	0:00:00	0:00:04	0:03:04	2:42:16		
heart-statlog	270	7	6	426	0:00:00	0:00:00	0:00:31	1:20:05			
liver	345	7	0	448	0:00:00	0:00:01	0:01:57	8:59:39			
mammographic	830	3	2	127	0:00:00	0:00:00	0:00:01	0:00:22	0:17:21	5:02:41	
mushroom	5644	0	22	98	0:00:00	0:00:00	0:00:06	0:01:29	0:22:16	5:57:51	
sick	2643	6	21	954	0:00:00	0:00:02	0:01:40	1:41:03			
vote	232	0	16	32	0:00:00	0:00:00	0:00:05	0:00:33	0:05:18	0:50:47	4:14:38
wine	178	13	0	732	0:00:00	0:00:01	0:00:19	0:16:58	14:36:11		

which is the same as the initial CBA algorithm described by Liu et al. (1998) except that the classification association rules are generated by means of the Apriori-TFP algorithm. We induced a set of rules with a single term that have a minimum support of 5% and a minimum confidence of 80%.
- CN2 (Clark and Niblett 1989): applies the separate-and-conquer strategy to induce an ordered rule set.
- 1R (Holte 1993): is a baseline learner that induces a classifier based on one single attribute.
- PART (Frank and Witten 1998): combines the decision tree approach and sequential covering by repeatedly generating partial decision trees.
- RIPPER (Cohen 1995): induces an ordered set of rules with a single term by combining covering with a reduced error pruning strategy. We used the WEKA implementation JRIP (Witten and Frank 1999).
- RISE (Domingos 1994): induces an unordered set of rules with a single term with the help of the conquering-without-separating strategy.
- SL^2 (Rückert and De Raedt 2008): induces a single DNF rule by a Stochastic Local Search algorithm. The number of terms in the DNF rule is minimized through internal cross-validation. We used the '$-l$' option of SL^2 to minimize the number of literals in the DNF rule.

The algorithms were evaluated by executing ten runs of a 10-fold cross-validation experiment on the twelve datasets used before. The same folds were used for all algorithms. Since CBA and SL^2 only operate on nominal features, we performed a simple frequency-based discretization to replace the continuously-valued features by ten new Boolean features, as previously done by Rückert (Rückert and De Raedt 2008). The other algorithms were run without discretization. In all datasets, examples with missing values were removed.

4.1 Performance comparison

Table 7 shows the average accuracy and standard deviation of the ten runs on the datasets. Differences between EXPLORE and the other algorithms were tested by a paired t-test at the 1% significance level in accordance with the study of Rückert (Rückert and De Raedt 2008). Results are marked with a filled circle "•" if EXPLORE performed significantly better than another algorithm and are marked with an open circle "○" if EXPLORE performed significantly worse. Our results show that there are only few datasets where another algorithm outperforms EXPLORE. Interestingly, for the diabetes and the liver datasets EXPLORE performs significantly better than all the other algorithms.

To compare the algorithms over multiple datasets we used a Friedman test with a conservative Bonferroni-Dunn post-hoc test as recently suggested by Demšar (2006). The Friedman test showed that there is a statistically significant difference in accuracy among the algorithms ($p = 0.006$) with EXPLORE having the highest average rank. In Fig. 3 the ranks of all the algorithms are presented together with the critical distance (marked interval) as defined by the Bonferroni-Dunn test. The results show that 1R, CN2, SL^2, and CBA perform significantly worse that EXPLORE overall, while C4.5R, PART, RISE, and RIPPER are comparable. Note that the average accuracy of C4.5R is lower than that of EXPLORE on all the datasets but the difference is not large enough to result in a rank outside the critical distance (cf. Table 7).

Table 7 Average accuracy (SD) of the rule learners

Dataset	C4.5R	CBA	CN2	1R	PART	RIPPER	RISE	SL^2	EXPLORE
australian	85.1 (0.5)	85.3 (0.6)	80.8 (0.8)●	85.5 (0.0)	84.4 (0.8)●	85.4 (0.8)	83.0 (0.5)●	85.2 (0.4)	85.2 (0.4)
breast-cancer	70.8 (1.8)	73.2 (3.0)	67.3 (1.7)●	68.4 (1.3)●	71.6 (1.9)	71.4 (1.3)	73.4 (1.0)○	69.1 (1.4)●	72.0 (0.9)
breast-w	95.4 (0.4)	94.8 (0.3)●	95.4 (0.6)●	91.7 (0.4)●	94.2 (0.4)●	94.2 (0.8)●	96.9 (0.3)○	94.3 (0.2)●	95.9 (0.4)
diabetes	72.9 (0.9)●	60.7 (1.9)●	71.3 (0.9)●	72.4 (0.8)●	73.7 (1.1)●	74.3 (0.6)●	71.8 (0.8)●	72.1 (0.7)●	75.8 (0.3)
glass(G2)	92.5 (0.8)●	85.4 (4.7)●	93.7 (0.8)	90.1 (0.3)●	92.5 (1.1)●	91.2 (1.0)●	94.5 (0.9)	87.5 (2.1)●	93.8 (0.7)
heart-statlog	79.2 (1.4)	81.7 (1.1)○	75.6 (1.7)●	71.3 (1.5)●	77.3 (1.3)●	79.7 (2.6)	79.2 (1.0)	70.8 (1.9)●	79.9 (1.4)
liver	63.9 (1.5)●	57.4 (1.5)●	64.0 (2.7)●	55.1 (1.7)●	64.4 (2.1)●	65.7 (1.2)●	63.9 (2.3)●	61.5 (2.9)●	68.1 (0.9)
mammographic	83.2 (0.4)	85.3 (0.4)○	77.2 (0.7)●	82.8 (0.0)	81.7 (0.7)●	83.6 (0.5)	78.7 (0.5)●	82.3 (1.0)	83.5 (0.7)
mushroom	100.0 (0.0)	99.9 (0.0)●	100.0 (0.0)	98.4 (0.0)	100.0 (0.0)	100.0 (0.0	100.0 (0.0)	100.0 (0.0)	100.0 (0.0)
sick	97.8 (0.1)	80.3 (10.0)●	97.9 (0.1)	95.3 (0.1)●	98.2 (0.2)○	98.0 (0.2)	97.6 (0.1)●	96.9 (0.2)●	97.8 (0.0)
vote	95.5 (0.4)●	90.6 (3.9)●	95.0 (0.5)●	97.0 (0.0)○	95.6 (0.4)●	96.7 (0.3)○	95.9 (0.4)●	96.8 (0.4)○	96.3 (0.1)
wine	93.7 (0.7)	86.7 (2.7)●	93.3 (1.3)	86.4 (1.3)●	94.0 (1.1)	92.0 (1.0)●	96.8 (0.9)○	92.6 (0.6)●	94.1 (0.5)
average	85.8 (0.8)	81.8 (2.5)	84.3 (1.0)	82.9 (0.6)	85.6 (0.9)	86.0 (0.9)	86.0 (0.7)	84.1 (1.0)	86.9 (0.5)

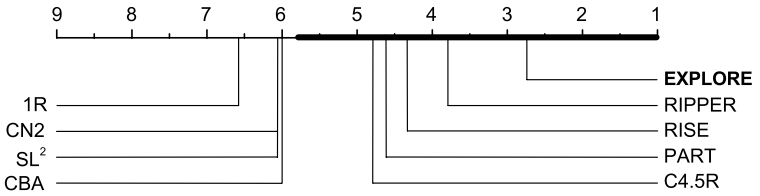

Fig. 3 Comparison of EXPLORE against the other algorithms with the Bonferroni-Dunn test. All algorithms with ranks outside the marked interval are significantly different ($p < 0.05$) from EXPLORE

4.2 Classifier size

The comprehensibility of a classifier strongly depends on its size. We compared the number of terms (Table 8) and the number of literals (Table 9) induced by EXPLORE and by the other rule learners. For the algorithms that generate a set of rules with a single term, we counted the total number of terms and literals in the set.

All algorithms induce classifiers with more terms and more literals than EXPLORE, except for SL^2 which generates a smaller number of terms. In particular, C4.5R, CBA, CN2, PART, and RISE generate classifiers with more literals. For example, for the mammographic dataset CN2 induces on average 283.9 literals, while EXPLORE uses only 3 literals and achieves a higher accuracy (cf. Table 7). Furthermore, the results on the mushroom dataset show that although all algorithms, except CBA and 1R, are able to find a classifier with 100% accuracy, only RIPPER and EXPLORE consistently find the least complex classifier containing 6 literals.

5 Discussion

In this study we show that our exhaustive approach to rule learning gives, on average, substantially smaller classifiers than those of eight rule learners, while securing comparable or even better performance. Although, it is interesting to see that on some datasets the other algorithms are able to induce rules with an accuracy comparable to exhaustive search in a fraction of the execution time, our study also shows that they can result in suboptimal solutions. This lower performance can probably be attributed to overfitting and/or searching a limited space. Our results indicate that EXPLORE is not very sensitive to overfitting, since only for the vote dataset it induced a longer rule than RIPPER and SL^2, with a lower performance. Interestingly, RISE performs better than EXPLORE on the breast-cancer and breast-w datasets, but with much more complex classifiers than those produced by all other algorithms. By default RISE combines all rules that are induced, including rules that win only a few or even no examples in the training set. RISE can induce a more comprehensible classifier if a threshold is used on the number of training examples won by a rule, but this could negatively affect the performance. Recently, Rückert has shown that the SL^2 algorithm was able to obtain a similar level of accuracy as state-of-the-art rule learners, with smaller DNF rules (Rückert and De Raedt 2008). According to our study the DNF rules induced by SL^2 based on stochastic local search are still larger than the DNF rules induced by EXPLORE utilizing exhaustive search and have a lower accuracy on most datasets. The necessary discretization step for SL^2 to handle continuously-valued features may have negatively affected its results. However, in the breast-cancer dataset all features were nominal and still EXPLORE performed better with respect to classifier size and inductive performance.

Table 8 Average number of terms (SD) of the rule learners

Dataset	C4.5R	CBA	CN2	1R	PART	RIPPER	RISE	SL^2	EXPLORE
australian	15.3 (0.4)●	27.3 (0.4)●	31.8 (0.7)●	2.0 (0.0)●	30.3 (1.7)●	3.2 (0.6)●	256.7 (3.5)●	1.0 (0.0)○	1.5 (0.2)
breast-cancer	8.8 (0.8)●	31.0 (1.0)●	54.0 (1.0)●	11.7 (0.7)●	16.8 (1.2)●	2.2 (0.3)○	115.0 (2.5)●	1.4 (0.2)○	2.5 (0.2)
breast-w	16.1 (0.3)●	21.0 (1.0)●	11.4 (0.2)●	10.0 (0.0)●	9.4 (0.8)●	11.7 (0.3)●	126.4 (4.1)●	1.9 (0.3)○	3.6 (0.2)
diabetes	18.5 (0.6)●	2.3 (0.5)●	49.5 (0.7)●	7.6 (0.4)●	6.7 (0.4)●	2.8 (0.3)●	384.0 (2.0)●	1.2 (0.1)○	1.6 (0.0)
glass(G2)	9.8 (0.2)●	11.3 (0.5)●	4.6 (0.2)●	2.0 (0.0)	3.9 (0.2)●	2.3 (0.4)	43.4 (2.1)●	1.9 (0.2)	2.1 (0.1)
heart-statlog	12.2 (0.2)●	37.3 (0.5)●	15.7 (0.3)●	2.0 (0.0)	16.4 (0.7)●	3.2 (0.4)●	90.1 (1.2)●	1.2 (0.1)○	2.0 (0.0)
liver	14.7 (0.5)●	1.0 (0.1)○	29.0 (0.6)●	10.9 (0.6)●	7.0 (0.6)●	3.4 (0.3)●	183.2 (1.4)●	2.6 (0.4)●	2.0 (0.0)
mammographic	8.7 (0.2)●	11.6 (0.4)●	96.2 (2.0)●	2.0 (0.0)○	12.1 (1.3)●	2.3 (0.1)	228.0 (5.2)●	1.9 (0.2)○	2.3 (0.1)
mushroom	11.6 (0.2)●	15.0 (0.0)	9.0 (0.0)	9.0 (0.0)	10.2 (0.3)●	5.0 (0.0)	13.8 (0.5)●	2.0 (0.1)○	5.0 (0.0)
sick	17.0 (0.3)●	35.2 (5.2)●	17.1 (0.6)●	4.0 (0.3)●	14.5 (0.8)●	5.9 (0.5)●	422.3 (11.5)●	2.1 (0.2)●	1.0 (0.0)
vote	4.2 (0.3)●	11.2 (0.8)●	8.5 (0.2)●	2.0 (0.0)○	4.2 (0.3)●	1.1 (0.1)○	25.0 (0.6)●	1.3 (0.1)○	2.2 (0.1)
wine	10.6 (0.2)●	14.1 (0.6)●	3.8 (0.2)●	2.6 (0.1)●	2.3 (0.1)	3.1 (0.1)●	19.7 (1.4)●	1.1 (0.1)○	2.3 (0.1)
average	12.3 (0.3)	18.2 (0.9)	27.6 (0.6)	5.5 (0.2)	11.1 (0.7)	3.8 (0.3)	159.0 (3.0)	1.6 (0.2)	2.3 (0.1)

Table 9 Average number of literals (SD) of the rule learners

Dataset	C4.5R	CBA	CN2	1R	PART	RIPPER	RISE	SL^2	EXPLORE
australian	40.2 (1.4)●	84.4 (1.4)●	99.2 (1.1)●	2.0 (0.0)○	75.1 (5.8)●	7.2 (1.8)●	2881.0 (44.1)●	2.0 (0.2)○	2.3 (0.3)
breast-cancer	19.4 (2.4)●	92.1 (3.2)●	124.8 (2.3)●	11.7 (0.7)●	35.1 (2.5)●	4.3 (0.6)○	797.5 (16.6)●	5.0 (1.6)	4.9 (0.5)
breast-w	19.8 (0.5)●	51.0 (2.0)●	29.2 (0.6)●	10.0 (0.0)●	10.9 (0.8)●	16.0 (1.2)●	1137.5 (37.1)●	15.5 (1.8)●	4.4 (0.3)
diabetes	36.8 (1.9)●	2.3 (0.5)	152.9 (1.1)●	7.6 (0.4)●	14.9 (1.4)●	7.8 (1.1)●	3071.8 (16.3)●	6.4 (1.3)●	2.4 (0.1)
glass(G2)	15.0 (0.5)●	11.7 (0.5)●	9.7 (0.5)●	2.0 (0.0)○	6.9 (0.3)●	4.4 (1.0)●	390.2 (19.0)●	16.7 (2.1)●	2.3 (0.2)
heart-statlog	29.2 (0.6)●	112.9 (1.6)●	46.5 (0.9)●	2.0 (0.0)○	51.1 (2.0)●	7.0 (1.0)●	1027.1 (15.1)●	4.2 (2.2)	3.7 (0.1)
liver	30.6 (1.2)●	1.0 (0.1)○	88.6 (1.4)●	10.9 (0.6)●	19.7 (1.6)●	8.9 (1.1)●	1099.2 (8.6)●	37.1 (7.1)●	2.9 (0.1)
mammographic	16.1 (0.4)●	26.7 (1.1)●	283.9 (5.3)●	2.0 (0.0)○	25.2 (3.7)●	3.9 (0.2)●	1084.4 (23.1)●	5.2 (0.9)●	3.0 (0.2)
mushroom	18.0 (0.4)●	32.0 (0.1)●	15.1 (0.1)	9.0 (0.0)	14.0 (0.2)●	6.0 (0.0)	172.9 (7.2)●	9.1 (0.3)●	6.0 (0.0)
sick	36.0 (0.9)●	80.9 (12.3)●	46.3 (1.6)●	4.0 (0.3)	50.6 (3.8)●	22.1 (2.1)●	10864.2 (306.7)●	18.8 (4.1)●	3.8 (0.0)
vote	8.1 (0.9)●	30.9 (2.4)●	15.8 (0.2)●	2.0 (0.0)○	5.7 (0.5)●	1.4 (0.3)○	260.9 (9.1)●	2.0 (0.3)○	4.2 (0.1)
wine	14.3 (0.3)●	14.1 (0.6)●	7.7 (0.4)●	2.6 (0.1)○	5.2 (0.2)●	4.9 (0.2)●	256.2 (18.7)●	8.3 (0.9)●	3.3 (0.1)
average	23.6 (0.9)	45.0 (2.1)	76.6 (1.3)	5.5 (0.2)	26.2 (1.9)	7.8 (0.9)	1920.2 (43.5)	10.9 (1.9)	3.6 (0.2)

Our findings contradict the paper of Quinlan and Cameron-Jones, which suggests that massive search reduces inductive performance (Quinlan and Cameron-Jones 1995). They called this phenomenon oversearching and studied it empirically by varying the amounts of search and comparing the performance of the different classifiers that were induced. Their hypothesis is that the more rules are evaluated the greater the chances of finding a fluke rule, i.e., a rule that accidentally fits the data well but does not represent a real pattern. Yet, we agree with Segal who states that this could also be the result of a faulty evaluation function which results in overfitting of the data (Segal 1997). Exhaustive and massive search algorithms have to limit the depth of the search to avoid loss of generalizability. In EXPLORE we use a validation set to determine the maximum length of the DNF rule.

We were able to shorten the execution time of EXPLORE significantly by incorporating several new techniques that greatly reduce the search space. Firstly, the application of the subsumption principle considerably reduces the number of values, as compared to taking the class-boundary approach as proposed by Fayyad and Irani (1992). Our new approach selects relevant values based on the type of operator ($>, \leq$) in the literal. In an earlier study a subsumption approach was used to select relevant literals (Lavrač et al. 1999). In addition, we here show that there is a direct relationship between the relational operator and the subset of relevant values in the literals. We believe this finding can be important for other algorithms as well to reduce the search space without loss of performance. Secondly, we apply a branch-and-bound technique to assure that rules are not generated and evaluated if they cannot attain a higher performance than the currently best performing rule. We defined novel bounding rules to induce DNF rules. Other algorithms, such as OPUS, have also used branch-and-bound to reduce the search space but for conjunctive rules only. We show that the branch-and-bound approach is not limited to accuracy optimization but can also be applied to problems in which sensitivity needs to be optimized with a minimum constraint on specificity. For this latter type of optimization, the user-defined constraint can strongly reduce the search space and shorten the execution time.

6 Future research

The current study is a starting point for a number of interesting research possibilities:

Firstly, EXPLORE is designed for binary classification, but we will extend EXPLORE to the multi-class scenario in the near future. The conventional way to handle this problem is to decompose the multi-class problem into a series of two-class problems and construct several binary classifiers, e.g., by the one-against-all method. The method used by CN2 to induce an unordered rule set can possibly be applied (Clark and Boswell 1991).

Secondly, in EXPLORE we use a validation set to determine the maximum length of the DNF rule. Other approaches can be applied for regularization in an exhaustive search, e.g. internal cross-validation instead of hold out. We need to study the effect of the regularization method on the results.

Thirdly, EXPLORE is able to learn DNF rules under one or more performance constraints. Learning under constraints is a form of cost-sensitive learning, and it would be interesting to compare the performance of EXPLORE in this respect with wrapper techniques and other cost-sensitive algorithms.

Fourthly, in this study only relatively small datasets from the UCI data repository have been used in the experiments because exhaustive search is not feasible in much larger problems. Yet, there are several possibilities to reduce the computation time of EXPLORE. Heuristics can be applied to reduce the search space. For example, one of the two possible relational operators for each continuous-valued attribute could be preselected instead of

considering both relational operators for each instantiation. Of course, such heuristics may negatively affect classification performance. Another option is to incorporate expert knowledge to guide the search. For example, based on prior knowledge some features can be made obligatory, relevant values can be specified by a range, or rules can be defined as a starting point for induction. The expert may also predefine a minimum value of the sensitivity by increasing the *TPbound* beforehand and then let EXPLORE optimize sensitivity with a constraint on the specificity. This can improve the speed of the search considerably because a higher bound at the start of the induction reduces the number of potential rules. Another advantage of the use of expert knowledge is that this may improve the comprehensibility of the resulting rule. Comprehensibility is generally greater for smaller rules and for rules that contain features that are easily interpretable for the end-user. Finally, the ever-increasing processor speed and the application of grid computing will allow exhaustive search on increasingly larger problems. The systematic rule generation of EXPLORE is eminently fit for parallel computation at various levels in the algorithm. A distributed version of EXPLORE that runs on a cluster of computers is under development.

7 Conclusions

In this study we present the EXPLORE algorithm, a new exhaustive search algorithm for finding a DNF rule. We describe a novel branch-and-bound approach for DNF rule learning, which reduces the execution time considerably. Also, we propose an operator-dependent version of the subsumption principle that can strongly reduce the number of relevant values and allows exhaustive search of much larger problems than previously possible. A comparison with eight rule learners showed that exhaustive search often results in more accurate and, in particular, less complex classifiers, and is to be preferred over non-exhaustive search if computationally feasible.

Acknowledgements We would like to thank Peter Clark, Tim Niblett, Ross Quinlan, Frank Coenen, Pedro Domingos and Ulrich Rückert for making available their learning systems. Furthermore, we gratefully acknowledge the developers of the WEKA environment.

Appendix A: Symbol table

Symbol	Definition
b	best DNF rule
$C(r, X)$	constraints of rule r on examples X
$fo_{i,j}$	feature-operator in literal j of term i
FO	list of feature-operators
FP_i	number of false positives of term i
$FPbound$	bound on the number of false positives
L	tuple of literals
$l_{i,j}$	literal j of term i
m	number of terms
n	rule length
$P(r, X)$	performance of rule r on example set X

r	DNF rule
T	tuple of term sizes
$TPbest$	number of true positives of best rule
TP_i	number of true positives of term i
$TP_{i,j}$	number of true positives of literal j in term i
$TPmax_i$	maximum number of true positives in term i
$TPbound_i$	bound on the number of true positives in term i
V	value lists of all feature-operators
$V(fo)$	list of values of feature-operator fo
$v_{i,j}$	value in literal j of term i
x	labeled example from set X

References

Andrews, G. E., & Eriksson, K. (2004). *Integer partitions*. New York: Cambridge University Press.

Bayardo, R. J. (1997). Brute-force mining of high-confidence classification rules. In *Knowledge discovery and datamining conference (KDD)* (pp. 123–126).

Clark, P., & Boswell, R. (1991). Rule induction with CN2: some recent improvements. In *Proceedings of the fifth European conference on machine learning (EWSL-91)* (pp. 151–163).

Clark, P., & Niblett, T. (1989). The CN2 induction algorithm. *Machine Learning*, *3*, 261–283.

Coenen, F. (2004). LUCS KDD implementation of CBA (classification based on associations). http://www.csc.liv.ac.uk1/~frans/kdd/software/cmar/cba.html. Accessed 8 August 2009.

Cohen, W. W. (1995). Fast effective rule induction. In *12th international conference on machine learning* (pp. 115–123).

Demšar, J. (2006). Statistical comparisons of classifiers over multiple data sets. *Journal of Machine Learning Research*, *7*, 1–30.

Dietterich, T. (1995). Overfitting and undercomputing in machine learning. *Computing Surveys*, *27*, 326–327.

Domingos, P. (1994). The RISE system: conquering without separating. In *6th IEEE international conference on tools with artificial intelligence* (pp. 704–707).

Domingos, P. (1999). Metacost: a general method for making classifiers cost-sensitive. In *5th international conference on knowledge discovery and data mining* (pp. 155–164).

Elkan, C. (2001). The foundations of cost-sensitive learning. In *7th international joint conference on artificial intelligence* (pp. 973–978).

Fayyad, U., & Irani, K. (1992). On the handling of continuous-valued attributes in decision tree generation. *Machine Learning*, *8*, 87–102.

Frank, E., & Witten, I. H. (1998). Generating accurate rule sets without global optimization. In *15th international conference on machine learning* (pp. 144–151).

Galen, R., & Gambino, S. (1975). *Beyond normality: the predictive value and efficiency of medical diagnosis*. New York: Wiley.

Holte, R. (1993). Very simple classification rules perform well on most commonly used datasets. *Machine Learning*, *11*, 63–90.

Kors, J. A., & Hoffmann, A. L. (1997). Induction of decision rules that fulfil user-specified performance requirements. *Pattern Recognition Letters*, *19*, 1187–1195.

Lavrač, N., Gamberger, D., & Jovanoski, V. (1999). A study on the relevance for learning in deductive databases. *Journal of Logic Programming*, *40*, 215–249.

Liu, B., Hsu, W., & Ma, Y. (1998). Integrating classification and association rule mining. In *Knowledge discovery and data mining* (pp. 80–86).

Michalski, R., Mozetic, I., Hong, J., & Lavrac, N. (1986). The multi-purpose incremental learning system aq15 and its testing application to three medical domains. In *5th national conference on artificial intelligence (AAAI-86)* (pp. 1041–1045).

Mitchell, T. (1997). *Machine learning*. New York: McGraw Hill.

Murphy, P., & Aha, D. (1994). UCI repository of machine learning databases. department of information and computer science. university of California, Irvine. http://www.ics.uci.edu/~mlearn/mlrepository.html Accessed 8 August 2009.

Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning*, *1*, 81–106.

Quinlan, J. R. (1992). *C4.5: programs for machine learning*. San Mateo: Morgan Kaufmann.

Quinlan, J. R., & Cameron-Jones, R. M. (1995). Oversearching and layered search in empirical learning. In *14th international joint conference on artificial intelligence* (pp. 1019–1024).

Rückert, U., & De Raedt, L. (2008). An experimental evaluation of simplicity in rule learning. *Artificial Intelligence*, *172*, 19–28.

Segal, R. (1997). *Machine learning as massive search* Dissertation, University of Washington.

Segal, R., & Etzioni, O. (1994). Learning decision lists using homogeneous rules. In *12th national conference on artificial intelligence (AAAI-94)* (pp. 619–625).

Viaene, S., & Dedene, G. (2005). Cost-sensitive learning and decision making revisited. *European Journal of Operational Research*, *166*(1), 212–220.

Webb, G. I. (1995). OPUS: an efficient admissible algorithm for unordered search. *Journal of Artificial Intelligence*, *3*, 431–465.

Weiss, S., Galen, R., & Tadepalli, P. (1990). Maximizing the predictive value of production rules. *Artificial Intelligence*, *45*, 47–71.

Witten, I. H., & Frank, E. (1999). *Data mining: practical machine learning tools and techniques with Java implementations*. San Mateo: Morgan Kaufmann.

Yin, X., & Han, J. (2003). CPAR: classification based on predictive association rules. In *SIAM international conference on data mining (SDM'03)* (pp. 331–335).

Zoghbi, A., & Stojmenovic, I. (1998). Fast algorithms for generating integer partitions. *International Journal of Computer Mathematics*, *70*, 319–332.