Advances in Data Analysis and Classification
https://doi.org/10.1007/s11634-023-00553-7

REGULAR ARTICLE

On the efficient implementation of classification rule learning

Michael Rapp[1] · **Johannes Fürnkranz**[2] · **Eyke Hüllermeier**[1]

Received: 2 June 2022 / Revised: 4 April 2023 / Accepted: 10 July 2023

Abstract
Rule learning methods have a long history of active research in the machine learning community. They are not only a common choice in applications that demand human-interpretable classification models but have also been shown to achieve state-of-the-art performance when used in ensemble methods. Unfortunately, only little information can be found in the literature about the various implementation details that are crucial for the efficient induction of rule-based models. This work provides a detailed discussion of algorithmic concepts and approximations that enable applying rule learning techniques to large amounts of data. To demonstrate the advantages and limitations of these individual concepts in a series of experiments, we rely on BOOMER—a flexible and publicly available implementation for the efficient induction of gradient boosted single- or multi-label classification rules.

Keywords Classification rules · Multi-label classification · Gradient boosting · Large-scale machine learning · Parallelization

Mathematics Subject Classification 68T09 · 62RO7

✉ Michael Rapp
michael.rapp@ifi.lmu.de

Johannes Fürnkranz
juffi@faw.jku.at

Eyke Hüllermeier
eyke@ifi.lmu.de

[1] Chair of Artificial Intelligence and Machine Learning, Ludwig-Maximilians-Universität München, Munich, Germany

[2] Institute for Application-oriented Knowledge Processing, Johannes Kepler University Linz, Linz, Austria

Published online: 27 July 2023

1 Introduction

Due to the increasing access to computational resources and the availability of large amounts of data in our digitalized world, machine learning methods are gaining importance and are nowadays used for many diverse applications. Among the wide variety of topics addressed by research on this growing and multidisciplinary field, the development of classification systems, which can provide categorical predictions for given data examples, belongs to the most established and well-known machine learning disciplines. This includes *concept learning* problems, where each example belongs to one out of two predefined classes. Based on the properties of an example, which we refer to as *attributes* or *features*, a classification method should be able to identify the class to which a particular example belongs. Many existing approaches to classification problems are based on the principles of *supervised learning*, where a predictive *model* is derived from given *training examples* for which the true classes are known. Besides *statistical* machine learning methods (see, e.g., Friedrich et al. 2022), *symbolic* approaches, which rely on symbolic descriptions to represent learned concepts, have a long history of active research (see, e.g., (Langley 1996; Mitchell 1997), for textbooks on the topic). Especially in safety-critical applications, where unexpected behavior may lead to life-threatening situations or economic loss, there is a need for interpretable machine learning models that can be analyzed and verified by human experts. As they capture knowledge about a problem domain in terms of simple conditional statements that humans can easily explore, understand and edit (see, e.g., Vojíř and Kliegr 2020), symbolic classification methods are often considered to meet these requirements. Consequently, they have received widespread attention in the literature on *interpretable machine learning* (see, e.g., Du et al. 2019; Murdoch et al. 2019; Molnar et al. 2020, for surveys on the topic).

In symbolical learning, *rule learning* algorithms are among the best-known and widely adopted approaches (see, e.g., Fürnkranz et al. 2012, for a textbook on the topic). A rule-based model is organized in terms of simple *if-then* statements referred to as *rules*. Each rule consists of one or several conditional clauses that compare the feature values of given examples to predetermined constants, which are well-suited for discrimination between different classes. If a rule's conditions are satisfied by an example, the rule is responsible for providing a prediction for one of the available classes. Rule-based models are closely related to *decision trees*, where the tests for an example's feature values are organized in a hierarchical, flowchart-like structure. For this reason, many techniques from decision tree learning can be transferred to rule learning approaches and vice versa. However, rule-based models provide additional flexibility when it comes to selecting rules that should be included in a model. Every decision tree can be transformed into an equivalent set of non-overlapping rules, where each path from the tree's root node to one of its leaves is considered an individual rule. However, not all rule-based models can be encoded as a tree-like structure. Consequently, rule-based models can be considered a more general concept class than decision trees (Rivest 1987). They are often preferred in applications that demand interpretability because decision trees tend to become quite complex in noisy domains due to their restriction to non-overlapping branches (Boström 1995). Moreover, identical sub-trees often occur within a single decision tree due to the fragmentation of

the example space that results from separation into non-overlapping regions (Pagallo and Haussler 1990). Rule-based models have also received attention for their ability to make the decision-making process of complex statistical methods transparent. Existing approaches either convert entire black-box models into rule-based representations (e.g., Zilke et al. 2016) or make use of rules to provide local explanations of their predictions (e.g., Ribeiro et al. 2016). This illustrates another attractive property of rule models. Whereas decision trees or rule sets can be viewed as global models, individual rules provide local explanations for the examples they apply to (Fürnkranz 2005; Lakkaraju et al. 2016). In addition, the locality of rules can be considered an advantage when used in *ensemble methods*. Similar to tree-based ensemble methods, such as Random Forests (Breiman 2001) or Gradient Boosted Decision Trees (GBDT) (Chen and Guestrin 2016; Ke et al. 2017), which combine the predictions of several trees, rule-based ensemble methods like ENDER (Dembczyński et al. 2010) or BOOMER (Rapp et al. 2020) achieve strong predictive results by employing a large number of overlapping rules. However, instead of combining several global models, the latter allow for a more efficient use of computational resources by learning more rules for regions where accurate predictions are difficult and fewer rules for regions that are easier to model. Such behavior has also been found to be beneficial in terms of computational efficiency by the authors of LightGBM (Ke et al. 2017). Based on prior work by Shi (2007), their GBDT method uses a "best-first" strategy, where only a few branches of a decision tree—which can be viewed as individual rules—are grown to the full extent, whereas less computational effort is put into others.

1.1 Motivation and goals

Despite the large number of publications on rule learning methods, only little information can be found about how such algorithms should be implemented at a low level. Rather than elaborating on algorithmic details that are important for a computationally efficient implementation, existing work on the topic primarily focuses on high-level concepts and ideas that aim to improve the effectiveness of different learning approaches in terms of predictive accuracy. With some notable exceptions, which we highlight in this paper, this also applies to the literature on decision tree learning. This is surprising because the increasing access to large amounts of data results in a growing demand for machine learning algorithms that can be applied to vast numbers of training examples. This requirement is also illustrated by the success and widespread adoption of highly scalable symbolic learning methods like XGBoost (Chen and Guestrin 2016) or LightGBM (Ke et al. 2017) that put great emphasis on computational efficiency.

This work sheds some light on the internals of some of the most efficient rule learning algorithms available today. First and foremost, this refers to the idea of using a so-called "pre-sorted" search algorithm for the construction of individual rules. In addition, we elaborate on extensions of this fundamental principle that allow dealing with nominal attributes and missing feature values natively. In contrast to most statistical machine learning approaches, including artificial neural networks, the ability to handle such data naturally, without any pre-processing techniques, is an advantage of

symbolic learning methods. We also discuss how sparsity in the feature values of training examples, which is a common characteristic of many datasets, can be exploited to further reduce computational costs. In domains with many numerical features, the ability to exploit sparsity in the feature values does not provide any significant advantages in terms of scalability. As an alternative, we also investigate a histogram-based search algorithm that generalizes existing ideas from decision tree learning to rule-based models. Finally, we elaborate on different possibilities to use parallelization to further speed up the training of predictive models. Even though most principles in this paper originate from existing work on rule and decision tree learning, we provide a unified view of these concepts and present them in great detail. Our discussion is complemented by a series of experimental studies that demonstrate the advantages and disadvantages of the individual optimizations and approximations examined in this work.

For our experiments, we rely on BOOMER (Rapp et al. 2020), a recently proposed and publicly available rule learning method.[1] Rapp (2021) provides a technical overview of the algorithm's capabilities. Even though this particular algorithm relies on the gradient boosting framework for the induction of rules, all principles and ideas discussed in this work apply to different types of rule learning methods as well. Furthermore, the BOOMER algorithm was designed with the particularities of *multi-label classification* in mind. This problem domain, where examples must not necessarily belong to a single class, includes binary classification as a special case. Not being restricted to single-class problems allows for a more general investigation of the algorithmic concepts discussed in this work.

1.2 Outline

In the following section, we start by discussing the preliminaries of the present work. First of all, this includes a definition of the classification problems we address in this work in Sect. 2.1. Then, a discussion of algorithmic concepts that different rule learning algorithms have in common and the particularities of the rule learning algorithm that serves as a basis for our experiments is given in Sects. 2.2, 2.3 and 2.4. Afterward, we discuss different aspects of the efficient induction of rules in each of the following sections:

- In Sect. 3, we illustrate the principles of the pre-sorted search algorithm employed by many existing rule learning methods. Its subsections are devoted to different extensions of this basic algorithm. This includes the possibility to exploit sparsity among the feature values in Sect. 3.2 and means to deal with nominal attributes and missing values in Sects. 3.3 and 3.4, respectively.
- As an alternative to the pre-sorted search algorithm in Sect. 3, a histogram-based approach is presented in Sect. 4. The individual subsections of this section focus on different aspects of this particular approximation technique.

[1] The BOOMER algorithm is publicly available at https://github.com/mrapp-ke/Boomer.

- Sect. 5 is concerned with the use of multi-threading to speed up the pre-sorted or histogram-based search algorithm. Section 5.1 considers the induction of single-label rules, as used in binary classification. The induction of multi-label rules, which is unique to the multi-label classification setting, is discussed in Sect. 5.2.

Each section mentioned above includes an experimental evaluation of the algorithmic aspects it focuses on. In Sect. 6, we draw conclusions from the experimental results and provide an outlook on ideas that are out of the scope of this work but may be used to further improve the scalability of rule learning methods.

2 Preliminaries

In this section, we provide a definition of multi-label classification problems, as well as an overview on the datasets and evaluation measures that we use for our experiments. We also discuss the basic structure most rule learning approaches have in common and recapitulate on the learning algorithm that we utilize in the remainder of this work.

2.1 Problem definition

In machine learning, classification systems are concerned with the assignment of data examples to classes. Algorithms aimed at binary or multi-class classification should be capable of assigning individual examples to one out of two or several mutually exclusive classes. Supervised learning approaches derive a predictive model from a limited set of labeled training examples for which the true classes are known. A good model should generalize beyond the provided observations such that it can be used to deliver predictions for yet unseen examples. Multi-label classification can be considered a generalization of traditional learning problems, where a single example may be associated with several class labels simultaneously. For example, in text classification, a single text document may belong to multiple topics. Hence, the goal of a multi-label classifier is to assign a given example to a subset of predefined labels. We specify the labels associated with the training examples in the form of a *label matrix* $Y \in \{0, 1\}^{N \times K}$, where each element $y_{nk} \in Y$ indicates whether the k-th label is relevant (1) or irrelevant (0) to the n-th example. If only a single label is available, i.e., if $K = 1$, a multi-label problem of this form simplifies to the task of predicting for either one of two classes, as necessary in binary classification.

In this work, we deal with structured data, where each example can be represented by a real-valued *feature vector* $\boldsymbol{x}_n = (x_{n1}, \ldots, x_{nL})$ that assigns constant *feature values* x_{nl} to numerical or nominal attributes A_l. Similar to the label matrix, we use a *feature matrix* $X \in \mathbb{R}^{NL}$ to specify the feature values that correspond to individual examples and attributes. A value corresponding to a numerical attribute may be any positive or negative real number. In contrast, the values of nominal attributes are restricted to an unordered set of categorical values. They are usually encoded by assigning a unique value to each available category. In this work, we also discuss means to deal with missing feature values, i.e., datasets where individual elements of an example's feature vector are unspecified.

2.2 Classification rules

In the remainder of this work, we are concerned with the construction of predictive models

$$R = (r_1, \ldots, r_T), \tag{1}$$

which consist of a predefined number of T rules. A model of this kind is typically assembled by following a sequential procedure, where one rule is added after the other (cf. Sect. 2.3). In accordance with existing rule learning literature, we denote an individual rule as

$$r_t : head \leftarrow body, \tag{2}$$

where the *body* is a conjunction of several conditions that specify the examples to which a rule applies and the *head* provides a deterministic or probabilistic prediction for these covered examples. Each condition that may be contained by a rule's body refers to one of the attributes in a dataset. It compares the value of an example for the corresponding attribute to a constant using a relational operator, such as $=$ and $\neq$, if the attribute is nominal, or $\leq$ and $>$, if the attribute is numerical.

2.3 Sequential model assemblage

Rule models are typically assembled by following a sequential procedure. As shown in Algorithm 1, more rules are added to a model until a certain stopping criterion is met, e.g., if a predefined model size has been reached. By learning one rule after the other, information about previous rules can be considered for learning subsequent ones. For example, in *separate-and-conquer* learning (Fürnkranz 1999), only examples that have not been covered by previous rules are taken into account when learning a new one. Similarly, *weighted covering* algorithms (Weiss and Indurkhya 2000; Gamberger and Lavrač 2000) weigh individual examples differently, depending on how often they have been covered yet. Without loss of generality, we refer to the information that is taken into account for inducing a single rule as *label space statistics*. Even though their exact notion depends on a particular rule learning approach at hand, they incorporate information about the true class labels of individual training examples and the corresponding predictions provided by previously induced rules. Traditional rule learning methods like RIPPER (Cohen 1995) typically characterize the predictions of rules in terms of confusion matrix elements, such as true positives, false positives, etc. In contrast, boosting algorithms like *ENDER* (Dembczyński et al. 2010), *BOOMER* (Rapp et al. 2020) or *isotonic boosting classification rules* (Conde et al. 2021) employ *gradients* and *Hessians* that guide the construction of rules (cf. Sect. 2.4). In both cases, the induction of a new rule (cf. FIND_RULE in Algorithm 1) requires the label space statistics to be updated, as it alters a model's predictions for the available training examples.

Algorithm 1 Sequential assemblage of a rule model

input: Feature matrix X, label matrix Y
$S =$ initialize label space statistics w.r.t. Y
$t = 1$
while no stopping criterion is met **do**
    $\boldsymbol{w}_t =$ determine weight of each example
    $r_t =$ FIND_RULE $(X, Y, \boldsymbol{w}_t, S)$
    $t = t + 1$
end while
return list of rules $R = (r_1, r_2, \dots)$

2.4 Top-down rule induction

As outlined by Hüllermeier et al. (2020), the construction of a single rule requires enumerating potential candidates for the rule's body and determining a corresponding head, as well as an estimate of the rule's quality, for each one of them.

2.4.1 Enumeration of rule bodies

Many rule learning algorithms employ a greedy top-down search, also referred to as *top-down hill-climbing*, to search for potential bodies (Fürnkranz et al. 2012). As outlined in Algorithm 2, it starts with an empty body that is iteratively refined by adding new conditions. The addition of new conditions results in fewer examples being covered, i.e., the rule is successively tailored to a subset of the available training examples. For each candidate body and depending on the head that is constructed for it, the quality of the predictions provided for the covered examples is assessed in terms of a numerical score $q \in \mathbb{R}$ that allows comparing different candidates to each other. The best candidate is selected at each iteration, and the possible refinements that result from adding a condition to the respective body are considered next. The search algorithm stops as soon as the quality of a rule cannot be improved by adding additional conditions to its body.

When following the procedure outlined above, refinements of a rule are selected greedily, i.e., the search focuses on a single refinement at each iteration. A *beam search* that explores a fixed number of alternatives, rather than focusing on a single refinement, may help overcome the search myopia that results from such a greedy hill-climbing approach (Fürnkranz et al. 2012). Alternatively, *branch-and-bound* algorithms (e.g., Boley et al. 2021) that rely on theoretical guarantees to prune the search space can be used to construct certifiably optimal rules. Nevertheless, we focus on greedy rule induction in the following. In particular, we elaborate on how the refinements of a rule can be evaluated efficiently (cf. FIND_REFINEMENT in Algorithm 2; an implementation is given in Algorithm 3), as this operation has the most significant impact on the computational costs of a greedy rule learner. Per its design, such a method can be implemented much more efficiently than branch-and-bound algorithms, approaches based on association rule mining (e.g., Lakkaraju et al. 2016), or methods that extract

rules from decision trees (e.g., Friedman and Popescu 2008; Bénard et al. 2021). Consequently, they are particularly well-suited for constructing large rule ensembles.

Algorithm 2 Recursive top-down search for a single rule (FIND_RULE)

input: Feature matrix X, label matrix Y, weights of training examples $\boldsymbol{w}$, label space statistics S, current rule r and its quality q (both optional), indicator function I (all examples are considered as covered if omitted)
best refinement $r^* = \emptyset$, best quality $q^* = q$
$\boldsymbol{s}$ = aggregate statistics of covered examples with non-zero weight
for all attributes A_l to be considered **do**
    $\boldsymbol{x}_l$ = retrieve vector with (sorted) feature values for A_l from X
    refinement r_l, its quality q_l = FIND_REFINEMENT $(\boldsymbol{x}_l, q^*, I, \boldsymbol{w}, S, \boldsymbol{s})$
    if q_l better than q^* **then**
        update best refinement $r^* = r_l$ and its quality $q^* = q_l$
    end if
end for
if $r^* \neq \emptyset$ **then**
    apply best refinement r^* to current rule r
    update I by marking all examples that satisfy r^* as covered
    return FIND_RULE $(X, Y, \boldsymbol{w}, S, r, q^*, I)$
end if
update statistics S w.r.t. ground truth Y and updated rule r
return updated rule r

2.4.2 Construction of rule heads

A corresponding head must be constructed for each candidate body considered during training. As a rule only provides predictions for examples it covers, its head should be tailored to the covered training examples. Compared to traditional classification settings, where the head of a rule is obliged to predict one of the available classes, a rule may provide predictions for a single label or several ones in multi-label classification. Following the terminology by Loza Mencía et al. (2018), we refer to rules that predict for single or multiple labels as *single-* or *multi-label rules*, respectively. Depending on the type of label space statistics used by a particular rule learning method, the construction of rule heads and the assessment of their quality may differ. Heuristic rule learning approaches like RIPPER (Cohen 1995) assess the quality of candidate rules in terms of a heuristic function based on confusion matrices. In contrast, boosting methods, such as the BOOMER (Rapp et al. 2020) algorithm used in this work, aim to optimize a continuous and differentiable (surrogate) loss function $\ell : \{0, 1\}^K \times \mathbb{R}^K \rightarrow \mathbb{R}_+$. It compares the probabilistic predictions provided by a model for K labels to binary ground truth labels. In such a case, the label space statistics for individual examples $\boldsymbol{x}_n$ consist of a *gradient vector* $\boldsymbol{g}_n$ and a *Hessian matrix* H_n. The elements of the gradient vector $\boldsymbol{g}_n = (g_{ni})_{1 \leq i \leq K}$ correspond to the first-order partial derivative of ℓ with respect to a model's predictions for one of the K labels. Accordingly, the

second-order partial derivatives form the Hessian matrix $H_n = \left(h_{nij}\right)_{1 \leq i,j \leq K}$. The individual gradients and Hessians are formally defined as

$$g_{in} = \frac{\partial \ell}{\partial \hat{p}_{ni}}\left(\boldsymbol{y}_n, \hat{\boldsymbol{p}}_n^{(t-1)}\right) \quad \text{and} \quad h_{ijn} = \frac{\partial \ell}{\partial \hat{p}_{ni} \partial \hat{p}_{nj}}\left(\boldsymbol{y}_n, \hat{\boldsymbol{p}}_n^{(t-1)}\right), \tag{3}$$

where $\boldsymbol{y}_n$ denotes the ground truth labels for the n-th training example and $\hat{\boldsymbol{p}}_n^{(t)}$ specifies the corresponding predictions of a model after t rules have been learned. In a multi-label setting one has to distinguish between *(label-wise) decomposable* and *non-decomposable* loss functions. Whereas the latter result in non-zero Hessians for each pair of label and therefore require to take interactions between labels into account, the former can be optimized more easily, as the gradient and Hessians for different labels are independent of each other (Dembczyński et al. 2012).

Regardless of the type of statistics used by a particular rule learning approach, constructing rule heads and evaluating their quality requires aggregating the statistics of several examples (cf., e.g., Algorithm 2, line 12). For this purpose, the aforementioned boosting approach sums up the gradient vectors and Hessian matrices that correspond to different examples in an element-wise manner. Furthermore, the methodology used to derive a rule's head from previously aggregated statistics depends on the type of rules to be learned (cf., FIND_HEAD in Algorithm 3). As the algorithmic concepts discussed in the following are independent of the methodology used for the construction of rule heads, we consider a detailed discussion of different approaches out of the scope of this work. Rapp et al. (2020) elaborate on how BOOMER derives the predictions of rules from gradient and Hessians in Section 4.1 of their work.

3 Pre-sorted search algorithm

For an efficient implementation of Algorithm 2, which aims at learning a single rule, an efficient evaluation of a rule's possible refinements is crucial (cf. Algorithm 2, FIND_REFINEMENT). Instead of evaluating each possible refinement in isolation, this can often be sped up by integrating the evaluation of multiple refinements into a single pass through the data. In this section, we discuss pre-sorting of examples as a way that works particularly well for numeric data. We will first discuss the base algorithm, and subsequently show how it can be extended to deal with sparse, nominal, and missing feature values.

3.1 Base algorithm for continuous attributes

A pre-sorted search algorithm sorts the available training examples by their values for individual attributes before training starts. Afterward, the examples are repeatedly processed in this predetermined order to build a model. This idea originates from early work on the efficient construction of decision trees (Mehta et al. 1996; Shafer et al. 1996). Due to the conceptual similarities between tree- and rule-based models, it can easily be generalized to rule learning methods. For example, it is used by

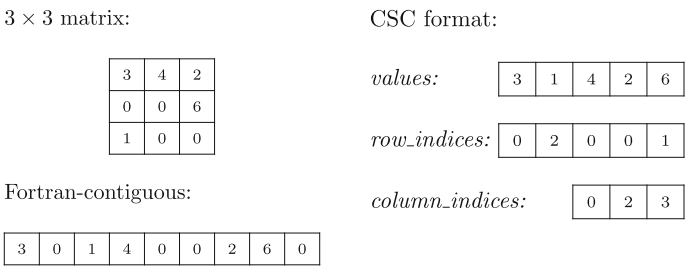


Fig. 1 Representation of a 3×3 matrix in the Fortran-contiguous and compressed sparse column (CSC) format. The former uses a single one-dimensional array to store all values in column-wise order, whereas the latter uses the following three arrays: (1) The array *values* stores all non-zero values in column-wise order. (2) For each value in *values*, *row_indices* stores the index of the corresponding row, starting at zero. (3) The i-th element in *column_indices* specifies the index of the first element in *values* and *row_indices* that belongs to the i-th column

"JRip", an implementation of RIPPER (Cohen 1995) that is part of the WEKA (Hall et al. 2009) project, or the implementations of SLIPPER (Cohen and Singer 1999) and RuleFit (Friedman and Popescu 2008) included in the "imodels" (Singh et al. 2021) package for interpretable models. Both rule- and tree-based learning approaches require enumerating the thresholds that may be used to make up nodes in a decision tree or conditions in a rule, respectively. These thresholds result from the feature values of the training examples, given in the form of a feature matrix as previously defined in Sect. 2.1. For each training example, it assigns a feature value to each of the available attributes. In the following, we restrict ourselves to numerical attributes before discussing means to deal with nominal attributes or missing feature values in Sects. 3.3 and 3.4, respectively.

When searching for the best condition that may be added to a rule, the available attributes are dealt with independently. For each attribute A_l to be considered by the algorithm, the thresholds that may be used by the first condition of a rule result from a vector of feature values $(x_{1l}, \ldots, x_{nl})$ that corresponds to the l-th column of the feature matrix. To facilitate column-wise access to the feature matrix, it should be given in the *Fortran-contiguous* memory layout (an example is given in Fig. 1). As different attributes are dealt with in isolation, we omit the index of the respective attribute for brevity. To enumerate the thresholds for a particular attribute, the elements in the corresponding column vector must be sorted in increasing order. For this purpose, we use a bijective permutation function $\tau : \mathbb{N}^+ \to \mathbb{N}^+$, where $\tau(i)$ specifies the index of the example that corresponds to the i-th element in the sorted vector

$$\left(x_{\tau(1)}, \ldots, x_{\tau(N)}\right) \text{ with } \tau(i) \leq \tau(i+1), \forall i \in [1, N). \tag{4}$$

An exemplary vector of sorted feature values, together with the corresponding thresholds, is shown in Fig. 2. Each of the thresholds is typically computed by averaging two adjacent feature values. Because these values do not change as additional conditions or rules should be learned, sorting the values that correspond to a particular attribute

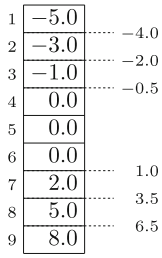


Fig. 2 Sorted vector of numerical feature values for a single attribute. The thresholds that result from averaging adjacent feature values are shown to the right

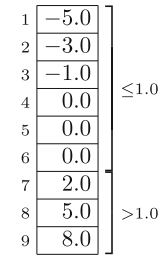


Fig. 3 Coverage of numerical conditions that can be created from a single threshold 1.0 using the $\leq$ or $>$ operator

is necessary only once during training, and previously sorted vectors can be kept in memory for repeated access.

If an existing rule should be refined by adding a condition to its body, only a subset of the feature values must be considered to make up potential thresholds. The subset corresponds to the examples that satisfy the existing conditions. We use an indicator function $I : \mathbb{N}^+ \rightarrow \{0, 1\}$ to check whether individual examples should be taken into account by the search algorithm:

$$I(n) = \begin{cases} 1 \text{ if example } \boldsymbol{x}_n \text{ is currently covered} \\ 0 \text{ otherwise.} \end{cases} \tag{5}$$

If an example is not covered, its feature value may not be used to make up thresholds for additional conditions. A data structure that helps to keep track of the covered examples efficiently, rather than comparing the feature values of each example to the existing conditions, is presented in Sect. 3.2 below. It also facilitates dealing with sparse feature values.

Algorithm 3 Pre-sorted search for the best refinement (FIND_REFINEMENT)

input: Vector of sorted feature attributes $\left(x_{\tau(n)}\right)_n^N$, quality of the current rule q, indicator function I, weights of training examples $\boldsymbol{w}$, statistics $S = \{(\boldsymbol{g}_n, H_n)\}_n^N$, globally aggregated statistics $\boldsymbol{s} = (\boldsymbol{g}, H)$ with $\boldsymbol{g} = \sum_n \left(I\left(n\right) w_n \boldsymbol{g}_n\right)$ and $H = \sum_n \left(I\left(n\right) w_n H_n\right)$
best refinement $r^* = \emptyset$, best quality $q^* = q$
for $i = 1$ **to** N **do**
    if $I\left(\tau\left(i\right)\right) = 1$ **and** $w_{\tau(i)} > 0$ **then**
        break
    end if
end for
initialize sum of gradients $\boldsymbol{g}' = \boldsymbol{g}_{\tau(i)}$ and Hessians $H' = H_{\tau(i)}$
for $j = i + 1$ **to** N **do**
    if $I\left(\tau\left(j\right)\right) = 1$ **and** $w_{\tau(j)} > 0$ **then**
        update sum $\boldsymbol{g}' = \boldsymbol{g}' + \boldsymbol{g}_{\tau(j)}$ and $H' = H' + H_{\tau(j)}$
        threshold $t = \text{avg}\left(x_{\tau(i)}, x_{\tau(j)}\right)$
        updated head $\hat{\boldsymbol{p}}'$, quality $q' = \text{FIND\_HEAD}\left(\boldsymbol{g}', H'\right)$
        if $q' < q^*$ **then**
            update refinement $r^* = \left\{t, \leq, \hat{\boldsymbol{p}}'\right\}$ and its quality $q^* = q'$
        end if
        $\hat{\boldsymbol{p}}', q' = \text{FIND\_HEAD}\left(\boldsymbol{g} - \boldsymbol{g}', H - H'\right)$ ▷ cf. Rapp et al (2020), Alg. 2
        if $q' < q^*$ **then**
            update refinement $r^* = \left\{t, >, \hat{\boldsymbol{p}}'\right\}$ and its quality $q^* = q'$
        end if
        $i = j$
    end if
end for
return best refinement r^*, its quality q^*

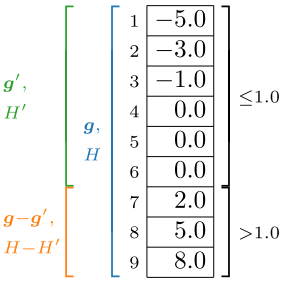


Fig. 4 Aggregation of statistics depending on the coverage of conditions that use the $\leq$ or $>$ operator and a single threshold 1.0. In case of the $>$ operator, the statistics are obtained by computing the difference (orange) between previously aggregated statistics (green), which correspond to already processed feature values, and globally aggregated statistics that are computed beforehand (blue) (color figure online)

3.1.1 Enumeration of conditions

As shown in Algorithm 3, the feature values that correspond to a particular attribute are processed in sorted order to enumerate the thresholds of potential conditions. When dealing with numerical attributes, the thresholds result from averaging adjacent feature values (cf. Algorithm 3, line 11). The calculation of thresholds is restricted to the feature values of examples that are covered according to the indicator function I and have non-zero weights according to a weight vector $\boldsymbol{w}$. The weights result from the application of an (optional) sampling method (cf. Algorithm 3, lines 4 and 10). When dealing with numerical attributes, each threshold can be used to form two distinct conditions, using the relational operator $\leq$ or $>$, respectively. As can be seen in Fig. 3, when a condition that uses the former operator is added to a rule, it results in all examples that correspond to the previously processed feature values being covered. In contrast, a condition that uses the latter operator covers all of the other examples.

3.1.2 Aggregation of statistics

As can be seen in the lines 13 and 17 of Algorithm 3, it is necessary to construct a head for each candidate rule that results from adding a new condition to a rule. In addition, the quality of the resulting rule must be assessed in terms of a numerical score. Both the predictions provided by a head and the estimated quality depend on the aggregated label space statistics of the covered examples. We exploit the fact that conditions using the $\leq$ operator, when evaluated in sorted order by increasing thresholds, are satisfied by a superset of the examples covered by the previous condition using the same operator but a smaller threshold. Processing the possible conditions in the aforementioned order enables the pre-sorted search algorithm to compute the aggregated statistics (corresponding to a vector of gradients $\boldsymbol{g}'$ and a matrix of Hessians H' in case of the BOOMER algorithm) incrementally (cf. Algorithm 3, line 12). For the efficient evaluation of conditions that use the $>$ operator, the search algorithm is provided with the statistics that result from the aggregation over all training examples that are currently covered and have a non-zero weight (denoted as $\boldsymbol{g}$ and H). The difference between the globally aggregated statistics and the previously aggregated ones ($\boldsymbol{g} - \boldsymbol{g}'$ and $H - H'$) yields the aggregated statistics of the examples covered by such a condition (cf. Algorithm 3, line 17). As the global aggregation of statistics does not depend on a particular attribute, this operation must be performed only once per rule, even when searching for a rule's best refinement across multiple attributes. Figure 4 provides an example of how the aggregated statistics are computed for the conditions that can be created from a single threshold.

3.2 Exploitation of feature sparsity

When dealing with training data where most feature values are equal to zero (or another predominant value), using a sparse representation of the feature matrix reduces the amount of memory required for storing its elements and facilitates the implementation of algorithms that can deal with such data in a computationally efficient way. In

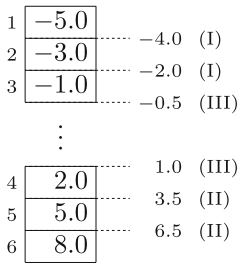


Fig. 5 Sparse representation of the vector of numerical features in Fig. 2, where values that are equal to zero are omitted. The thresholds that result from averaging adjacent feature values are shown to the right. The numbers in parentheses (I, II, III) specify the phases of the sparsity-aware search algorithm that are responsible for the evaluation of individual thresholds

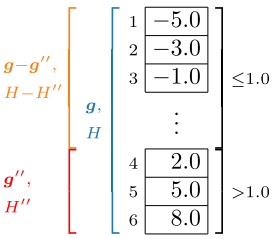


Fig. 6 Evaluation of conditions that separate examples with positive values from the remaining ones. The condition that uses the $\leq$ operator requires to compute the difference (orange) between the statistics of examples with positive values (red) and the globally aggregated ones (blue) (color figure online)

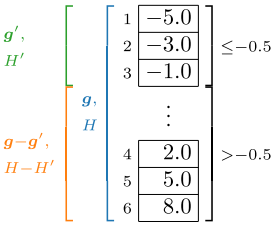


Fig. 7 Evaluation of conditions that separate examples with negative values from the remaining ones. The difference (orange) between the statistics of examples with negative values (green) and the globally aggregated ones (blue) is required in case of the $\leq$ operator (color figure online)

the following, we discuss a variant of the previously introduced pre-sorted search algorithm that allows to search for potential refinements of rules using both dense and sparse feature matrices. As shown experimentally in Sect. 3.5, the use of sparse input data, where feature values are provided in the *compressed sparse column* (CSC) format (see Fig. 1 for an example), may drastically reduce training times.

3.2.1 Enumeration of thresholds

When dealing with a sparse representation of the feature matrix, the sorted column vector for each attribute, which serves as a basis for enumerating possible thresholds, only contains non-zero feature values (an example is given in Fig. 5). On the one hand, because only non-zero values must be processed, this reduces the algorithm's computational complexity. However, on the other hand, the algorithm cannot identify the examples with zero feature values. Therefore, to enumerate all thresholds that result from a sparse vector, including those that result from examples with zero feature values, a "sparsity-aware" search algorithm must follow three phases:

- *Phase I:* It starts by processing the sorted feature values in increasing order as before. Traversal of the feature values must be stopped as soon as a positive value or a value equal to zero is encountered.
- *Phase II:* Afterward, it processes the sorted feature values in decreasing order until a negative value or a value equal to zero is encountered.
- *Phase III:* After all elements in a given vector have been processed, it is possible to deal with the thresholds that eventually result from examples with zero feature values. To determine whether such examples exist, the number of elements with non-zero weights processed so far can be compared to the total number of examples in a dataset or a sample thereof. If all available examples have already been processed, a single threshold can be formed by averaging the largest negative feature value and the smallest positive one that has been encountered in the previous phases. Otherwise, two thresholds between zero and each of these values can be made up.

An algorithm that follows the aforementioned procedure is not only able to deal with dense and sparse vector representations, but also enumerates all thresholds that are considered by Algorithm 3. However, if a large fraction of the feature values are equal to zero, it involves far less computational steps. The step-wise procedure that we use for the enumeration of thresholds resembles ideas that are used by XGBoost (Chen and Guestrin 2016) for the construction of decision trees from sparse feature values. However, the latter requires to traverse the values that correspond to each attribute twice, whereas our approach processes the values only once. In general, the proposed approach is not restricted to the space-efficient representation of zero values, but can omit explicit storage of any value that is predominant in the data. For simplicity, we restrict ourselves to the former.

3.2.2 Aggregation of statistics (phase I)

The first phase of the sparsity-aware search algorithm, where examples with negative feature values are processed, is identical to Algorithm 3. During this initial phase, the aggregated statistics of examples that satisfy potential conditions are obtained as illustrated in Fig. 4. If a condition uses the $\leq$ operator, the statistics of the examples it covers correspond to the previously aggregated statistics ($\boldsymbol{g}'$ and H') of already processed examples. To obtain the statistics of examples covered by a condition that uses the $>$ operator, the difference ($\boldsymbol{g} - \boldsymbol{g}'$ and $H - H'$) between the globally aggregated statistics ($\boldsymbol{g}$ and H) and the previously aggregated ones are computed. The first phase

ends as soon as an example with a positive or zero feature value is encountered. The statistics that have been aggregated until this point ($\boldsymbol{g}'$ and H') include all examples with negative feature values and are retained for later use during the third phase of the algorithm.

3.2.3 Aggregation of statistics (phase II)

The second phase, where examples with positive feature values are considered, follows the same principles as the previous phase. However, the examples are processed in decreasing order of the their respective feature values. Consequently, the incrementally aggregated statistics (denoted as $\boldsymbol{g}''$ and H'' to distinguish them from the variables used in the first phase) updated at each step correspond to the examples that cover a condition using the $>$ operator. To obtain the aggregated statistics of examples that satisfy a condition that uses the $\leq$ operator, the difference ($\boldsymbol{g} - \boldsymbol{g}''$ and $H - H''$) between the globally aggregated statistics and the previously aggregated ones must be computed. The end of the second phase is reached as soon as an example with a negative or zero feature value is encountered. The statistics aggregated during this phase ($\boldsymbol{g}''$ and H'') include the statistics of all examples with positive feature values. They are kept in memory for use during the third and final phase.

3.2.4 Aggregation of statistics (phase III)

After the second phase has finished, the algorithm is able to decide whether any examples with zero feature values, which are neither stored by a sparse representation of the feature values nor can explicitly be accessed by the rule induction algorithm, are available. This is the case if the sum of the weights of all examples processed until this point is smaller than the total sum of weights of all examples in a dataset or a sample thereof. In any case, it is necessary to evaluate potential conditions that separate the examples with positive feature values from the remaining ones (possibly including examples with zero feature values). As shown in Fig. 6, this requires considering two conditions with the operator $\leq$ and $>$, respectively. The statistics of examples that satisfy the latter correspond to the statistics aggregated during the algorithm's second phase ($\boldsymbol{g}''$ and H''). To obtain the statistics of examples that are covered by the former, the difference ($\boldsymbol{g} - \boldsymbol{g}''$ and $H - H''$) between the statistics aggregated during the second phase and the globally aggregated ones must be computed. In addition, if any examples with zero feature values are available, additional conditions using the operators $\leq$ and $>$ that separate the examples with negative feature values from the remaining ones must be considered. The statistics of examples that are covered by the former correspond to the statistics that have been aggregated during the first phase of the algorithm ($\boldsymbol{g}'$ and H'). In contrast, the statistics of examples that satisfy the latter must again be computed by taking the globally aggregated statistics into account. They calculate as the difference ($\boldsymbol{g} - \boldsymbol{g}'$ and $H - H'$) between the statistics corresponding to examples with negative feature values, which have been processed during the first phase, and the globally aggregated ones that have been computed beforehand. An example of how the aggregated statistics for the evaluation of these conditions are

obtained is given in Fig. 7. If no examples with zero feature values are available, the evaluation of additional conditions, as depicted in Fig. 7, can be omitted.

3.2.5 Keeping track of covered examples

Once the best condition among all available candidates has been added to a rule, it is necessary to keep track of the examples that are covered by the modified rule. This is crucial because additional conditions that may be added during later refinement iterations must only be created from the feature values of examples that satisfy the existing conditions. When dealing with a dense representation of feature values, as shown in Fig. 2, the feature values of all examples can directly be accessed. Keeping track of the covered examples is straightforward in this case. However, given a sparse representation, as shown in Fig. 5, it becomes a non-trivial task since the algorithm does not know the examples that come with zero feature values. To overcome this problem, we utilize a data structure suited to keep track of the covered examples in both cases, regardless of the feature representation used. It maintains a vector that stores a value for each example in a dataset, as well as an *indicator value*. If the value that corresponds to a certain example is equal to the indicator value, it is considered to be covered. This enables to answer queries to the indicator function $I(n)$, as defined in (5), in constant time by comparing the value of the n-th example to the indicator value. Initially, when a new rule does not contain any conditions yet, the indicator value and the values in the vector are all set to zero, i.e., all examples are considered to be covered by the rule. An example of such a data structure for nine examples that correspond to the feature values in Fig. 5 is shown in Fig. 8. Updating the data structure after a new condition has been found requires to take the range of examples it covers into account. If only examples with negative (or positive) feature values satisfy the respective condition, i.e., if the condition's threshold is less than (greater than or equal to) zero and it uses the $\leq$ ($>$) operator, the corresponding values of the proposed data structure can be updated directly. In such a case, the values of covered examples and the indicator value are set to the number of conditions that are currently contained in the rule's body, marking them as covered. If a condition is satisfied by examples with zero feature values, for which the corresponding indices are unknown, the values that correspond to the uncovered examples are updated instead by setting them to the current number of conditions. However, the indicator value remains unchanged, which renders the examples that correspond to the updated values uncovered, whereas examples with unmodified values remain covered if they already satisfy the previous conditions.

3.3 Dealing with nominal attributes

An advantage of rule learning algorithms is their ability to deal with nominal attributes by using operators like $=$ or $\neq$ for the conditions in a rule's body. This is in contrast to many statistical machine learning methods that cannot deal with nominal attributes directly. Instead, they require to apply preprocessing techniques to the data before training. Most commonly, *one-hot-encoding* is used to convert nominal attributes to

numerical ones. It replaces a single nominal attribute with a fixed number of discrete values with several binary attributes that specify for each of the original values whether it applies to an example or not. Such a conversion may drastically increase the number of attributes in a dataset and therefore can negatively affect the complexity of a learning task.

To deal with nominal attributes, we rely on the same principles used by the presorted search algorithm in the case of numerical attributes, including the ability to use sparse representations of feature values. In the case of a nominal attribute, the feature values associated with the individual training examples are not arbitrary real numbers but are limited to a predefined set of discrete values that do not necessarily correspond to a continuous range and possibly include negative values. As a result, the thresholds that potential conditions may use are not formed by averaging adjacent feature values but correspond to the discrete values associated with the available training examples. Two conditions must be evaluated for each of the values encountered by the algorithm in a sorted vector of feature values. As shown in Fig. 9, they use the operator $=$ and $\neq$, respectively. Whereas a condition that uses the former operator covers neighboring

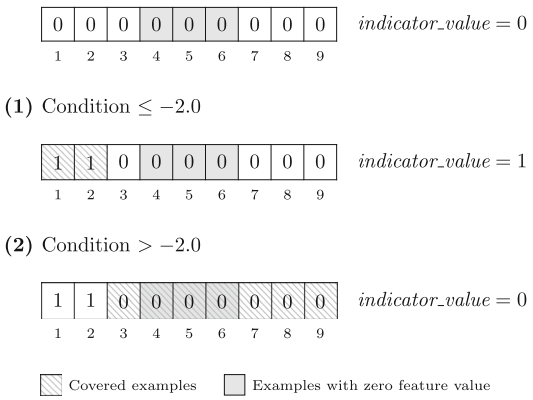
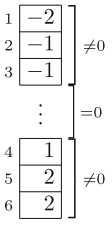


Fig. 8 Visualization of the data structure that is used to keep track of the examples that are covered by a rule. For each example (see Fig. 2 for their feature values), it stores a value that indicates whether the example is covered or not. An example is considered to be covered if its value is equal to an *indicator_value*. Initially, all examples are marked as covered (top). When a new condition is added to the rule, the data structure is updated by following one of the following strategies: (1) If the examples that satisfy the condition do not have zero feature values, the corresponding elements and the *indicator_value* are both set to the total number of conditions. (2) Otherwise, the elements that correspond to uncovered examples are updated, whereas the *indicator_value* remains unchanged

Fig. 9 Coverage of nominal conditions that can be created from a single threshold 0 using the $=$ or $\neq$ operator



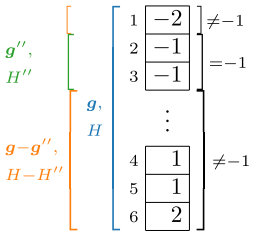


Fig. 10 Evaluation of nominal conditions that separate examples with a particular value from the remaining ones. The difference (orange) between the statistics of the covered examples (green) and the globally aggregated ones (blue) is used to evaluate conditions with the $\neq$ operator (color figure online)

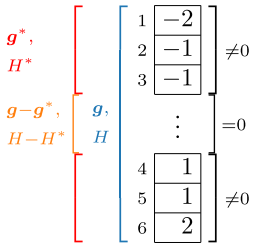


Fig. 11 Evaluation of nominal conditions that separate examples with zero values from the remaining ones. The statistics of the former result from the difference (orange) between the globally aggregated statistics (blue) and the ones of previously processed examples (red) (color figure online)

examples with the same value, the examples that satisfy a condition with the latter operator do not correspond to a continuous range in a sorted vector of feature values. This requires adjustments to the algorithm when it comes to the aggregating statistics that correspond to examples that are covered by nominal conditions.

3.3.1 Aggregation of statistics in the nominal case (phase I and II)

The algorithm follows the same order for processing the sorted feature values as outlined in Sect. 3.2 to facilitate the use of sparse feature representation when dealing with nominal attributes. At first, it processes the examples associated with negative feature values. Afterward, it evaluates the conditions that result from positive feature values, and finally, in a third phase, potential conditions with zero thresholds are considered. During the first and second phase, the statistics of examples with the same feature value are aggregated individually. In accordance with the notation in Fig. 10, we denote the aggregated statistics for different feature values as $\boldsymbol{g}'$, $\boldsymbol{g}''$, … and H', H'', …. This is in contrast to the aggregation of statistics in the case of numerical attributes, where the statistics of all examples with negative and positive feature values are aggregated. As illustrated in Fig. 10, the globally aggregated statistics ($\boldsymbol{g}$ and H), which are provided to the algorithm beforehand, are used to obtain the statistics corresponding to examples that satisfy conditions using the $\neq$ operator. This requires to compute the

difference between the globally aggregated statistics and the aggregated statistics of all examples associated with a particular discrete value.

3.3.2 Aggregation of statistics in the nominal case (phase III)

During the third phase of the algorithm, special treatment is required to evaluate conditions with zero thresholds if any examples with zero feature values are available. To determine whether such examples exist, the sum of the weights of all examples that have previously been processed in the first and second phases of the algorithm is compared to the weights of all examples in a dataset or a sample thereof, as described earlier. To obtain the aggregated statistics that correspond to the examples with zero feature values, the statistics $\boldsymbol{g}', \boldsymbol{g}'', \ldots$ and $H', H'', \ldots$ that have been computed during the previous phases must be aggregated. We denote the resulting accumulated statistics as $\boldsymbol{g}^* = \boldsymbol{g}' + \boldsymbol{g}'' + \ldots$ and $H^* = H' + H'' + \ldots$, respectively. As shown in Fig. 11, they correspond to the examples with non-zero feature values covered by a condition that uses the $\neq$ operator. To evaluate a condition that uses the $=$ operator and covers all examples with zero feature values, inaccessible by the algorithm when using a sparse feature representation, the difference between the globally aggregated statistics ($\boldsymbol{g}$ and H) and the accumulated ones are computed.

3.4 Support for missing feature values

The ability to deal with training data, where the feature values of individual examples are partly unknown, is a common requirement of many real-world classification problems. Therefore, we elaborate on ways to deal with unknown feature values in the following.

Different strategies for handling missing values can be found in the rule learning literature (see, e.g., Wohlrab and Fürnkranz 2011, for an overview). The algorithm, which is used for the experiments in this work, ignores examples with missing values when evaluating the conditions that can be added to a rule's body with respect to a certain attribute. Consequently, rules that contain conditions on an attribute A_l in their body can never be satisfied by examples $\boldsymbol{x}_n$ for which the feature value x_{nl} is missing. Other possible solutions include the opposite strategy, i.e., conditions on missing feature values are always satisfied, as well as the possibility to learn conditions that explicitly test for unknown values. Alternatively, some learning algorithms come with strategies to impute the values that are unknown for an example or ignore examples with missing values entirely.

Only minor adjustments are necessary to apply the previously presented search algorithm to an attribute for which some examples may lack a value. First of all, the examples that do not assign a value to the respective attribute are excluded from the sorted feature vector in (4) and therefore are ignored when enumerating the possible thresholds of conditions and aggregating the statistics of examples they cover. Instead, the algorithm keeps track of the examples that do not assign a value to a particular attribute in a separate data structure. When searching for possible conditions concerned with the respective attribute, the statistics of the examples that are known to lack a

value for the attribute must be subtracted from the globally aggregated statistics (previously denoted as $\boldsymbol{g}$ and H). This ensures that the statistics that are aggregated while processing a vector of sorted feature values, as well as the statistics that are computed as the difference between previously aggregated statistics and the globally aggregated ones, do not include examples with missing values, the considered refinements of a rule cannot cover.

3.5 Experimental evaluation

To evaluate the efficiency of the pre-sorted search algorithm employed by BOOMER and, in particular, the advantages that result from using its sparsity-aware variant, we applied the algorithm to several benchmark datasets (cf. Appendix A). For a complete picture, we included datasets with varying degrees of feature sparsity in the experimental study and considered both dense and sparse feature representations for the experiments. Due to the focus on computational efficiency and because the presented optimizations do not affect algorithmic behavior, the experimental results are mostly restricted to training times. However, we report the predictive performance achieved by relevant configurations of BOOMER in Appendix B.

3.5.1 Experimental setup

As we are interested in training efficiency, rather than predictive accuracy, we did not use any parameter tuning for the following experiments but used the BOOMER algorithm's default parameters instead. By default, the algorithm constructs models that consist of 1, 000 single-label rules. The search for rules is guided by a label-wise decomposable loss function, which serves as a surrogate for the Hamming loss (see, e.g., Gibaja and Ventura 2014, for a definition of the evaluation measures used in this work). Whenever a rule should be refined, it considers a subset of $\lfloor \log_2(L - 1) + 1 \rfloor$ random attributes. For the construction of an individual rule, all available training examples are taken into account rather than employing a sampling method. To deal with the nominal attributes that are included in some of the considered datasets, we relied on BOOMER's native support for this kind of attribute rather than applying one-hot-encoding to the data. The BOOMER algorithm's ability to utilize multiple computational threads was not used in the following experiments. A discussion of different possibilities to speed up training via multi-threading, accompanied by performance benchmarks, can be found in Sect. 5 below.

3.5.2 Experimental results

Table 1 provides an overview of the BOOMER algorithm's training times, averaged across the folds of a 10-fold cross validation, when applied to different datasets and using dense or sparse representations of feature values.[2] In addition, it also shows the relative speedup in training time that result from the exploitation of sparsity in

[2] For each experiment in this work we provided exclusive access to the following computational resources: AMD Ryzen 7 3800X (8 cores at 3.9 GHz) and 128 GB RAM.

Table 1 Average training times (in seconds) per cross validation fold on datasets with varying feature sparsity. The small numbers indicate the speedup that results from using sparse feature representations, compared to dense feature representations

Dataset	Feature sparsity	Feature representation		
		Dense	Sparse	
20NG	96.81	9.7	**1.1**	8.82
Bibtex	96.26	12.1	**1.9**	6.37
Birds	38.64	**0.6**	**0.6**	1.00
Bookmarks	94.16	308.4	**44.0**	7.01
Delicious	96.34	159.6	**20.0**	7.98
Emotions	0.33	**0.5**	**0.5**	1.00
Enron	91.60	1.0	**0.2**	5.00
EukaryoteGO	99.86	5.1	**0.4**	12.75
EukaryotePseAAC	43.37	6.5	**4.6**	1.41
EUR-Lex-SM	95.26	70.9	**9.3**	7.62
Image	0.22	**2.3**	**2.3**	1.00
IMDB	98.06	126.0	**12.2**	10.33
Langlog	81.38	0.9	**0.3**	3.00
Mediamill	0.00	**235.2**	235.5	1.00
Medical	99.32	0.9	**0.2**	4.50
Ohsumed	0.00	7.4	**0.9**	8.22
Reuters-K500	98.41	4.5	**1.0**	4.50
Scene	1.15	3.2	**3.0**	1.07
Slashdot	99.46	2.6	**0.2**	13.00
TMC2007	99.79	33.9	**1.3**	26.08
Yahoo-computers	99.62	19.6	**0.9**	21.78
Yahoo-reference	99.59	9.9	**0.6**	16.50
Yahoo-science	99.53	8.8	**0.5**	17.60
Yahoo-social	99.71	17.3	**0.9**	19.22
Yeast	0.00	2.7	**2.6**	1.04

The bold values are the best ones for each dataset/row

the feature space when using the latter type of feature representation.[3] As expected, when applied to datasets with low feature sparsity ("Emotions", "Image", "Mediamill", "Scene" and "Yeast"), the training times remain mostly unaffected by using sparse data structures. Notably, the use of sparse data structures does not harm training efficiency in these cases. On datasets with a feature sparsity between 30 and 50% ("Birds", "EukaryotePseAAC"), a minor speedup can be observed. On the remaining datasets, where at least 80% of the feature values are equal to zero, the sparsity-aware rule

[3] Given the duration d_1 and d_2 required by two competing approaches M_1 and M_2 for training, we measure their relative difference in training time as the ratio d_1/d_2.

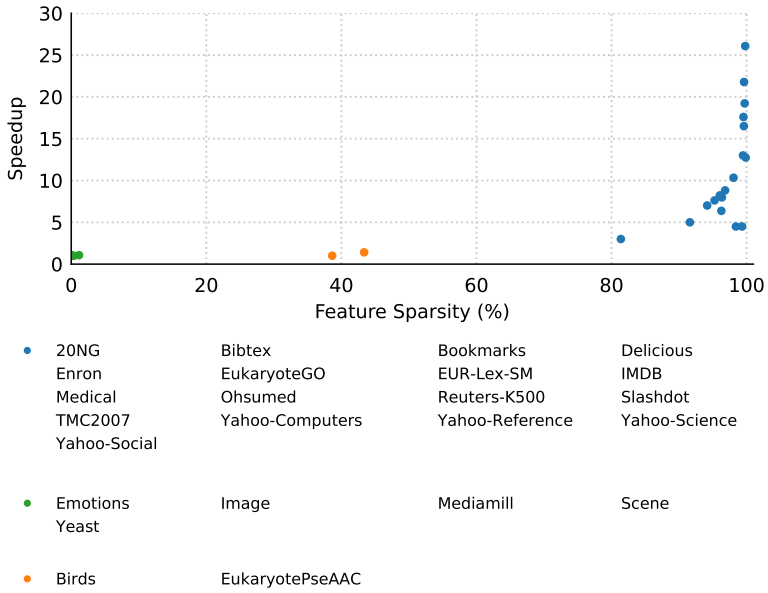


Fig. 12 The average speedup in training time per cross validation fold that results from a search algorithm that is capable of exploiting sparsity in the feature space, compared to the use of dense data structures. The considered datasets are grouped by small (green), medium (orange) and high (blue) feature sparsity (color figure online)

induction algorithm clearly outperforms the baseline and significantly reduces training time. This is also illustrated by the graphical representation of the experimental results in Fig. 12, where the speed up that results from using a sparse feature representation is related to the feature sparsity of the considered datasets.

4 Histogram-based rule induction

The exploitation of feature sparsity, as previously discussed in Sect. 3.2, helps reduce training times on many benchmark datasets that are used in this work, as they often come with high feature sparsity. However, it does not provide significant advantages on datasets with low feature sparsity. The BOOMER algorithm provides an alternative to the pre-sorted search algorithm introduced in Sect. 3 to efficiently deal with the latter type of datasets. It is based on assigning examples with similar values for a particular attribute to a predefined number of bins and using an aggregated representation of their corresponding label space statistics, referred to as *histograms*. Depending on how many bins are used, this approach drastically reduces the number of candidate rules the rule induction algorithm must consider. Histogram-based approaches have previously been used to deal with complex classification tasks in modern GBDT implementations, such as XGBoost (Chen and Guestrin 2016) or LightGBM (Ke et al. 2017). In the following, we discuss a generalization of the underlying concept, which has evolved from prior research on decision tree learning (Alsabti et al. 1998; Jin and Agrawal

2003; Li et al. 2007; Kamath et al. 2002), to rule learning methods and investigate its impact on predictive performance and training efficiency in an empirical study.

4.1 Assignment of examples to bins

A histogram-based rule induction algorithm requires grouping the available training examples into a predefined number of bins. Different approaches can principally be used to determine such a mapping (see, e.g., Kotsiantis and Kanellopoulos 2006, for a survey on existing techniques). We restrict ourselves to unsupervised binning methods, where the assignment is solely based on the feature values of the training examples. This is in contrast to supervised methods, such as the *weighted quantile sketch* approach that originates from the XGBoost algorithm (Chen and Guestrin 2016), where information about the true class labels of individual examples, or even their label space statistics, are taken into account. Compared to approaches that utilize the label space statistics map from examples to bins, unsupervised binning methods can usually be implemented more efficiently. This is because a mapping solely based on feature values remains unchanged for the entire training process, whereas the statistics for individual examples are subject to change and require adjusting the mapping whenever a model is refined.

4.1.1 Equal-width feature binning

The first binning method that we consider for our experiments is referred to as *equal-width binning*. This method, which is commonly used to discretize numerical feature values, is based on dividing the range of values for a particular attribute into equally-sized intervals, such that the absolute difference between the smallest and largest value in each bin are the same. Given a predefined number of bins B, the maximum difference between the values that are assigned to an interval calculates as

$$w = \frac{max - min}{B}, \tag{6}$$

where *min* and *max* denote the largest and smallest value in a bin, respectively. Based on the value w, a mapping $\sigma : \mathbb{R} \rightarrow \mathbb{N}^+$ from individual feature values x_n to the index of the corresponding bin can be obtained as

$$\sigma_{eq.\text{-}width}(x_n) = \min\left(\left\lfloor \frac{x_n - min}{w} \right\rfloor + 1, B\right). \tag{7}$$

4.1.2 Equal-frequency feature binning

Another well-known method to discretize numerical features considered in this work is *equal-frequency binning*. Unlike equal-width binning, which is supposed to result in bins with values close to each other, this particular discretization method aims to obtain bins that contain approximately the same number of values. The available examples

are first sorted in ascending order by their respective feature values to determine the bins for a particular attribute. This results in a sorted vector of feature values $\big(x_{\tau(1)}, \ldots, x_{\tau(N)}\big)$, where the permutation function $\tau(i)$ specifies the index of the example that corresponds to the i-th element in the sorted vector, as previously defined in (4). Afterward, the sorted values are divided into a predefined number of intervals, such that each bin contains the same number of values. Given an individual feature value x_n, the index of the corresponding bin calculates as

$$\sigma_{eq.\text{-}freq.}(x_n) = \lfloor \tau(n) - 1 \rfloor + 1. \tag{8}$$

In practice, examples with identical feature values should be prevented from being assigned to different bins. However, for reasons of brevity, this is omitted from the above formula.

4.1.3 Assignment of discrete values to bins

To handle datasets that do not only include numerical feature values, but also come with nominal attributes, we use an appropriate binning method to deal with the latter. It creates a bin for each discrete value encountered in the training data and assigns examples with identical values to the same bin.

4.2 Enumeration of thresholds

We denote the set of example indices that have been assigned to the b-th bin via a mapping function σ as

$$\mathcal{B}_b = \{n \in \{1, \ldots, N\} \mid \sigma(x_n) = b\}. \tag{9}$$

Given B bins previously created for a particular attribute, one can obtain $B - 1$ thresholds that the conditions of potential candidate rules may use. Depending on whether the $\leq$ or $>$ operator is used by a condition, the b-th threshold separates the examples that correspond to the bins $\mathcal{B}_1, \ldots, \mathcal{B}_b$ from the examples that have been assigned to the bins $\mathcal{B}_{b+1}, \ldots, \mathcal{B}_B$. The individual thresholds $t_1, \ldots, t_{B-1}$ calculate as the average of the largest and smallest feature value in two neighboring bins $\mathcal{B}_b$ and $\mathcal{B}_{b+1}$. Depending on the characteristics of the binning method at hand, some bins may remain empty. For the enumeration of potential thresholds, bins that are not associated with any examples should be ignored. When dealing with bins that have been created from nominal feature values, all examples in a particular bin have the same feature value. In such a case, the conditions in a rule's body may test for presence or absence of these B feature values.

4.3 Creation of histograms

When using unsupervised binning methods, the mapping of examples to bins and the thresholds resulting from individual bins must only be determined once during

training. They should be obtained when a particular attribute is considered by the rule induction algorithm for the first time and should be kept in memory for repeated access. In contrast, the histograms that serve as a basis for evaluating candidate rules must be created from scratch whenever a rule should be refined. As shown in Algorithm 3, they result from aggregating the label space statistics of examples that have been assigned to the same bin. Examples that do not satisfy the conditions that have previously been added to the body of a rule must be ignored. As defined in (5), we use an indicator function I to keep track of the examples that are covered by a rule. In addition, the extent to which the statistics of individual training examples contribute to a histogram depends on their respective weights. This enables the histogram-based search algorithm to use different samples of the available training examples to induce individual rules.

Algorithm 4 Creation of histograms from label space statistics

input: Bins $(\mathcal{B}_b)_b^B$, statistics $S = \{(\boldsymbol{g}_n, H_n)\}_n^N$, indicator function I, weights of training examples $\boldsymbol{w}$
initialize empty histogram $S' = \{(\boldsymbol{g}'_b, H'_b)\}_b^B$, where all elements of $\boldsymbol{g}'_b$ and H'_b are set to zero
for $n = 1$ **to** N **do**
    if $I(n) = 1$ **and** $w_n > 0$ **then**
        obtain bin index $b = \sigma(x_n)$
        update S' by setting $\boldsymbol{g}'_b = \boldsymbol{g}'_b + w_n \boldsymbol{g}_n$ and $H'_b = H'_b + w_n H_n$
    end if
end for
return histogram S'

4.4 Evaluation of refinements

When using a histogram-based search algorithm, evaluating candidate rules in terms of a given loss function follows the same principles as its pre-sorted counterpart in Algorithm 3. However, instead of taking the feature values of individual training examples into account for making up conditions that can be added to a rule's body, the conditions to be considered by the histogram-based algorithm result from the predetermined thresholds that correspond to the bins for a particular attribute. Even when an existing rule should be refined, i.e., when an existing rule covers only a subset of the training examples, the thresholds remain unchanged to increase the algorithm's efficiency. Similar to the pre-sorted rule induction algorithm, the histogram-based approach is based on incrementally aggregating the statistics of training examples that are covered by the considered refinements. However, instead of aggregating statistics at the level of individual training examples, it relies on the statistics that correspond to the individual bins of a histogram. For the efficient evaluation of conditions that use the $>$ operator in case of numerical attributes, or the $\neq$ operator in case of nominal attributes, the algorithm is provided with globally aggregated statistics that are determined beforehand and computes the difference between previously processed statistics

that correspond to individual bins and the globally aggregated ones as illustrated in Fig. 4. The statistics of examples with missing feature values are excluded from the globally aggregated statistics, as previously described in Sect. 3.4. In addition, the respective examples are ignored when determining the mapping to individual bins. Consequently, the histogram-based rule induction method can handle missing feature values.

4.5 Experimental evaluation

To compare the histogram-based rule induction algorithm to its pre-sorted counterpart, we investigated the predictive performance and training times of both approaches in an experimental study. We restricted our experiments to large datasets with many examples and low feature sparsity, as the histogram-based algorithm aims to reduce the time needed for training in such use cases. Among the considered benchmark datasets, only two datasets—"Mediamill" and "Nus-Wide cVLADplus"—meet these criteria (cf. Table 5 in the appendix).

4.5.1 Experimental setup

For our experiments, we configured the BOOMER algorithm in the same way as described in Sect. 3.5, i.e., we learned models of single-label rules that minimize a surrogate for the Hamming loss. The algorithm's ability to use multi-threading was again not used. It is elaborated on in Sect. 5 below. The main goal of our experiments was to investigate how the training time and predictive performance in terms of the Hamming loss, the subset 0/1 loss, and the example-wise F1-measure are affected by varying numbers of bins. Hence, we set the number of bins to be used by the histogram-based approach to 64, 32, 16, 8, or 4% of the distinct feature values available for a particular attribute. In addition, we also included a rather extreme setting, where the number of bins was limited to 8 bins. To assign the available training examples to the available bins, we further tested both binning methods supported by BOOMER, i.e., equal-width and equal-frequency binning.

4.5.2 Experimental results

Table 2 shows the predictive performances and training times that result from applying the pre-sorted and histogram-based algorithm to the considered datasets. It can be seen that the latter can reduce the time needed for training, regardless of the dataset and the configuration. As expected, the speedup in training time that results from the histogram-based approach increases when fewer bins are used. The equal-width binning method tends to be slightly more efficient than the equal-frequency method. This is most probably because the former does not require sorting the training examples and therefore comes with linear instead of logarithmic complexity. On the dataset "Mediamill" the use of equal-width binning reduces the training time up to a factor of 4.8, whereas the training algorithm finishes up to 22 times faster on the dataset "Nus-Wide-cVLADplus". Regardless of the number of bins, we observe a minor deterioration in

Table 2 Predictive performance of the pre-sorted and histogram-based rule induction algorithm in terms of Hamming loss, subset 0/1 loss and example-wise F1 measure, as well as the average training times (in seconds) per cross validation fold. The number of bins to be used by the histogram-based approach was set to 64, 32, 16, 8 or 4% of the distinct feature values for a particular attribute or to 8 bins

Dataset	Pre-sorted	Equal-frequency binning					
		64%	32%	16%	8%	4%	8 bins
Hamming loss							
Mediamill	**3.16**	3.22	3.23	3.22	3.23	3.22	3.24
Nus-Wide cVLADplus	**2.19**	2.23	2.23	2.23	2.23	2.23	2.24
Subset 0/1 loss							
Mediamill	**93.01**	93.89	93.93	93.84	93.89	93.89	93.99
Nus-Wide cVLADplus	**77.03**	77.52	77.53	77.54	77.52	77.55	77.55
Example-wise F1 measure							
Mediamill	**50.61**	49.67	49.67	49.75	49.69	49.65	49.27
Nus-Wide cVLADplus	**27.54**	25.52	25.53	25.57	25.54	25.57	25.10
Training time							
Mediamill	238.6	167.0	109.8	71.9	52.8	46.1	**39.6**
Nus-Wide cVLADplus	4623.7	760.3	641.1	514.4	402.3	281.9	**208.3**
Dataset	Pre-sorted	Equal-width binning					
		64%	32%	16%	8%	4%	8 bins
Hamming loss							
Mediamill	**3.16**	3.22	3.22	3.22	3.22	3.22	3.27
Nus-Wide cVLADplus	**2.19**	2.23	2.23	2.23	2.23	2.23	2.24
Subset 0/1 loss							
Mediamill	**93.01**	93.91	93.82	93.78	93.81	93.81	94.10
Nus-Wide cVLADplus	**77.03**	77.53	77.54	77.53	77.52	77.56	77.58
Example-wise F1 measure							
Mediamill	**50.61**	49.64	49.70	49.75	49.68	49.74	48.79
Nus-Wide cVLADplus	**27.54**	25.51	25.52	25.54	25.52	25.59	25.09
Training time							
Mediamill	238.6	106.9	75.9	58.6	50.1	45.8	**40.0**
Nus-Wide cVLADplus	4623.7	634.3	513.6	391.5	284.6	241.5	**206.7**

The bold values are the best ones for each dataset/row

predictive performance for all reported evaluation measures compared to the pre-sorted rule induction algorithm. Even though the performance of the histogram-based approach appears to be very resilient against a limitation of the available bins, the variant that is limited to 8 bins always comes with the most significant drop in predictive performance. On the considered datasets, the evaluation scores that are achieved by the equal-width and equal-frequency binning methods are close to each other and do not differ to a degree that is statistically significant.

5 Multi-threading

To utilize the multi-core architecture or hyper-threading capabilities of today's CPUs, one may consider to speed up the training of rule-based models by executing certain algorithmic aspects in parallel rather than sequentially. Unlike ensemble methods, where individual members are independent of each other, e.g., in Random Forests (Breiman 2001), most rule learning methods do not allow to construct individual rules in parallel due to the sequential nature of their training procedure, where each rule is built in the context of its predecessors. Instead, the following possibilities exist to parallelize computational steps that are involved in the induction of a single rule:

- *Multi-Threaded Evaluation of Refinement Candidates* The evaluation of conditions that can possibly be added to a rule's body requires enumerating the feature values of the training examples for each available attribute, aggregating the label space statistics of examples they cover, and computing the predictions and quality of the resulting candidates. Multi-threading may be used to evaluate the refinement candidates for different attributes in parallel.
- *Parallel Computation of Predictions and Quality Scores* For each candidate considered during the construction of a single rule, the predictions for different class labels and an estimate of their quality must be obtained. These operations are particularly costly if interactions between labels should be considered as is often the case in multi-label classification. In such a scenario, the parallelization of these operations across several labels may help reduce training times.
- *Distributed Update of Label Space Statistics* After a rule has been learned, the label space statistics of all examples it covers must be updated. The complexity of this operation depends on how many examples are covered and is affected by the number of labels for which a rule predicts. Moreover, the update becomes more costly if statistics are not only provided for individual labels but also for pairs of labels or even entire label sets. Depending on the methodology used by a particular rule learning approach, training times may be reduced by updating the statistics for different examples in parallel.

The benefits of using the aforementioned possibilities for parallelization heavily depend on the characteristics of a particular dataset and the learning algorithm. In some cases, the overhead of managing and synchronizing multiple threads outweighs the speedup that the parallel execution of computations may achieve. Consequently, the use of multi-threading may even have a negative effect on the time that is needed for training. To investigate the effects of multi-threading in an empirical study, we restrict ourselves to two common use cases: First, we investigate a setting where single-label rules are used to minimize a decomposable loss function. Second, we consider optimizing a non-decomposable loss function using multi-label rules that provide predictions all available labels. As the latter requires taking dependencies between labels into account, it is computationally more challenging to find the heads of candidate rules (cf. FIND_HEAD in Algorithm 3). Similar to Sect. 3.5 and 4.5, we use default settings for the remaining parameters and rely on the BOOMER algorithm's ability to use sparse feature representations, if appropriate according to Table 1.

5.1 The decomposable case

In the first experiment, we investigated how the time needed by the BOOMER algorithm for learning single-label rules is affected by the use of multi-threading. According to preliminary experiments, training efficiency is unlikely to benefit from updating the label space statistics in parallel in this particular setting. This is because single-label rules only affect the statistics that correspond to a single label. Due to the small costs of such a++n update operation, there is only a small potential for improvements. Similarly, we have observed that the benefits of using multiple threads for computing predictions and quality scores tend to be small in the decomposable case. During the first iteration of the rule induction algorithm, where candidate rules that contain a single condition in their body are considered, the algorithm must yet decide on a label to predict for. During this initial phase of rule construction, the aforementioned operations come with linear complexity. When evaluating the possible refinements of an existing rule, after the algorithm has decided for a particular label, they even reduce to constant-time operations that cannot be parallelized. Based on these findings, we restrict our study to the multi-threaded evaluation of refinement candidates across different attributes.

5.1.1 Experimental results

Table 3 reports the training times that result from using a single- or multi-threaded implementation to evaluate candidate rules. By default, BOOMER selects a random subset of the available attributes whenever a rule should be refined. This ensures that the resulting model consists of diverse rules that achieve high predictive accuracy in combination and results in a significant reduction of training time. With such a method for complexity reduction in place, the degree to which multi-threading can be expected to result in runtime improvements mostly depends on the feature sparsity of the training data. Whereas training can be three times faster on datasets with low feature sparsity, multi-threading tends to negatively affect training times if feature sparsity is very high. To investigate how runtimes are affected by the number of attributes that the rule induction algorithm must consider, we also conducted experiments with feature sampling disabled. In such a setting, the multi-threaded implementation outperforms the single-threaded baseline on all considered datasets. As shown in Fig. 13, the use of multi-threading has greater potential for significant speedups (up to a factor of 7) when the training algorithm must process more attributes.

5.2 The non-decomposable case

A key functionality of BOOMER is its capability to minimize non-decomposable loss functions. In addition to the experiments in Sect. 5.1, we investigate the use of multi-threading to speed up training in this particular scenario. As the training objective, we use a non-decomposable surrogate for the subset 0/1 loss. According to the findings by Rapp et al. (2020), the use of multi-label rules is crucial for the successful minimization of non-decomposable loss functions due to their ability to model

Table 3 Average training times (in seconds) per cross validation fold on different datasets when minimizing a decomposable loss function. The small numbers indicate the speedup that results from the use of multi-threading (using 8 threads) to evaluate the potential refinements of rules with respect to different attributes in parallel, compared to a single-threaded implementation

Dataset	Threads		
	1	8	
With Feature Sampling			
20NG	**1.1**	1.6	0.69
Bibtex	**1.9**	2.5	0.76
Birds	0.6	**0.3**	2.00
Bookmarks	44.0	**35.3**	1.25
Delicious	20.0	**19.6**	1.02
Emotions	0.5	**0.2**	2.50
Enron	**0.2**	0.3	0.67
EukaryoteGO	**0.4**	0.6	0.67
EukaryotePseAAC	6.5	**2.6**	2.50
EUR-Lex-SM	**9.3**	9.4	0.99
Image	2.3	**0.8**	2.88
IMDB	12.2	**12.0**	1.02
Langlog	**0.3**	0.4	0.75
Medical	**0.2**	0.3	0.67
Ohsumed	**0.9**	1.2	0.75
Reuters-K500	**1.0**	1.5	0.67
Scene	3.2	**1.0**	3.20
Slashdot	**0.2**	0.3	0.62
Yahoo-computers	**0.9**	1.4	0.64
Yahoo-reference	**0.6**	0.9	0.67
Yahoo-science	**0.5**	0.8	0.63
Yahoo-social	**0.9**	1.4	0.64
Yeast	2.7	**0.9**	3.00
Without Feature Sampling			
20NG	39.0	**14.6**	2.67
Bibtex	86.1	**20.2**	4.26
Birds	15.7	**2.8**	5.61
Bookmarks	3729.1	**1571.0**	2.37
Delicious	261.4	**129.9**	2.01
Emotions	4.5	**1.0**	4.50
Enron	11.7	**5.2**	2.25
EukaryoteGO	71.3	**62.2**	1.15
EukaryotePseAAC	258.0	**36.6**	7.05
EUR-Lex-SM	1677.0	**445.0**	3.77
Image	60.3	**9.7**	6.22

Table 3 (continued)

IMDB	500.0	**121.8**	4.11
Langlog	17.7	**4.5**	3.93
Medical	6.4	**5.4**	1.19
Ohsumed	40.7	**12.2**	3.34
Reuters-K500	36.2	**8.8**	4.11
Scene	83.4	**13.7**	6.09
Slashdot	21.0	**15.7**	1.34
Yahoo-computers	1080.3	**392.7**	2.75
Yahoo-reference	759.8	**282.7**	2.69
Yahoo-science	733.4	**274.0**	2.68
Yahoo-social	1608.5	**576.7**	2.79
Yeast	32.9	**5.3**	6.21

The bold values are the best ones for each dataset/row

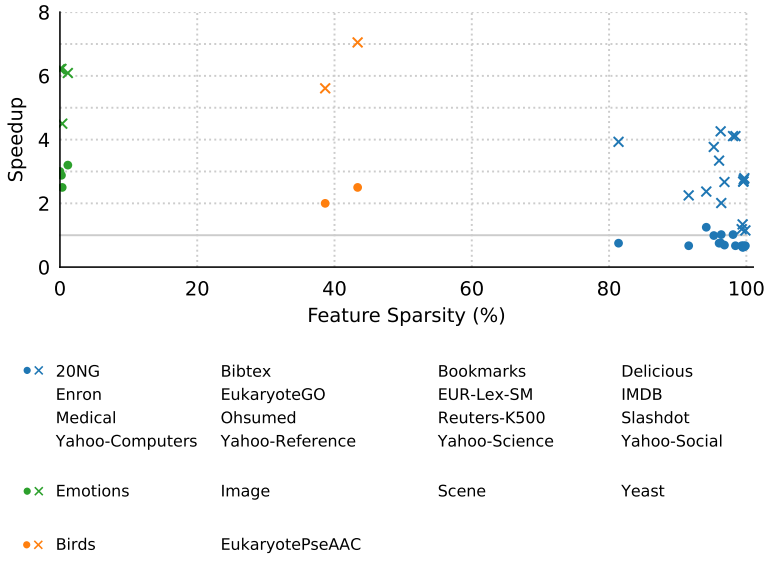


Fig. 13 The average speedup (or slowdown) in training time per cross validation fold that results from the use of multi-threading (using 8 threads) to evaluate the potential refinements of rules with respect to different attributes in parallel, compared to a single-threaded implementation. Regardless of whether feature sampling is used (circles) or not (crosses), the speedup mostly depends on whether a dataset has low (green), medium (orange) or high (blue) feature sparsity (color figure online)

dependencies between labels. We learn complete rules that provide predictions for all available labels in the following study to cater to these results. Calculating loss-minimizing predictions for different candidate rules requires solving a linear system in the non-decomposable case. Moreover, a matrix–vector multiplication must be performed to obtain an estimate of a candidate's quality. The BOOMER algorithm relies on the software libraries LAPACK (Anderson et al. 1999) and BLAS (Blackford et al.

Table 4 Average training times (in seconds) per cross validation fold on different datasets when minimizing a non-decomposable loss function. Three different configurations that use multi-threading (using 8 threads) to parallelize different aspects of the BOOMER algorithm are considered. Multi-threading can be used for linear algebra operations, updating the statistics of different examples in parallel or evaluating the potential refinements of rules across different attributes in parallel

Dataset	Linear algebra	Linear algebra & statistic update		Statistic update & refinements	
20NG	6.3	5.6	1.13	**4.1**	1.54
Bibtex	171.7	99.1	1.73	**86.4**	1.99
Birds	**32.8**	34.0	0.97	46.1	0.71
Emotions	**16.5**	17.5	0.94	28.2	0.59
Enron	5.8	4.6	1.26	**2.6**	2.23
EukaryoteGO	3.9	**2.5**	1.56	2.6	1.50
EukaryotePseAAC	**236.7**	237.7	1.00	335.4	0.71
EUR-Lex-SM	2483.8	2062.7	1.20	**1309.5**	1.90
Image	**71.7**	74.4	0.96	195.1	0.37
IMDB	80.5	48.3	1.67	**41.3**	1.95
Langlog	14.9	11.1	1.34	**6.6**	2.26
Medical	4.7	2.1	2.24	**1.9**	2.47
Ohsumed	6.3	5.5	1.15	**4.0**	1.58
Reuters-K500	130.2	104.1	1.25	**88.6**	1.47
Scene	**187.8**	189.8	0.99	519.7	0.36
Slashdot	2.9	**2.8**	1.04	3.5	0.83
Yahoo-computers	9.7	8.2	1.18	**7.7**	1.26
Yahoo-reference	7.5	6.3	1.19	**5.8**	1.29
Yahoo-science	8.5	7.2	1.18	**6.3**	1.35
Yahoo-social	13.0	9.7	1.34	**9.3**	1.40
Yeast	**87.4**	88.3	0.99	**128.7**	0.68

The bold values are the best ones for each dataset/row

2002) to implement these linear algebra operations. Multiple computational threads may be utilized to solve these operations. In addition, multi-threading can optionally be used to update the statistics that correspond to different examples in parallel whenever a new rule is added to a model. When dealing with a non-decomposable loss function, the number of statistics that must be maintained for each example grows exponentially with the number of available labels. Compared to the decomposable case, this makes the update operation more complex and offers potential for runtime improvements via parallelization. Finally, we also consider using multi-threading to parallelize the search for refinements across different attributes. As the evaluation of candidate rules involves the previously mentioned linear algebra operations, this particular parallelization strategy cannot be used in combination with the multi-threading capabilities offered by BLAS and LAPACK, which must be disabled to avoid problems that result from nested multi-threading. However, if the number of labels for which

a rule may predict is reasonably small and depending on the feature sparsity of a particular dataset, a parallel search for refinements may exhibit greater speedups than achievable by using multi-threaded linear algebra operations. As BOOMER comes with an approximation technique, referred to as *gradient-based label binning* (Rapp et al. 2021), which imposes an upper bound on the number of distinct predictions that may be provided by a rule, the former strategy for parallelization appears to be promising in many use cases.

5.2.1 Experimental results

The training times that result from the use of different multi-threading strategies in the non-decomposable case are shown in Table 4. Compared to a configuration where multiple computational threads are only used for linear algebra operations, the additional use of parallelization to update the statistics of different examples reduces training times on 15 out of the 21 considered datasets. Whether a speedup (up to a factor of 2) can be achieved primarily depends on the number of labels. On datasets with less than 20 labels ("Birds", "Emotions", "Image", "Scene" and "Yeast"), the overhead that is introduced by the additional use of multiple threads negatively affects the time needed for training. Similarly, the benefits of using multi-threading to evaluate possible refinements of rules across different attributes in parallel, rather than utilizing multiple threads for linear algebra operations, depend on the dataset. Unlike in the decomposable case, where this particular parallelization strategy is particularly efficient on datasets with small feature sparsity, the opposite can be observed in the non-decomposable setting. To understand this behavior, it is necessary to recall the principles of the sparsity-aware rule induction algorithm in Sect. 3.2. When dealing with datasets that come with sparse feature values, the number of candidate rules that must be considered by the algorithm drastically reduces. Therefore, compared to a dataset with small feature sparsity, the amount of training time spent on linear algebra operations is significantly smaller in such a case. Consequently, there is less potential to speed up these operations by using multi-threading. For this reason, multi-threading should be used to parallelize the search for refinements across multiple attributes on datasets with high feature sparsity (resulting in speedups up to a factor of 2.5). In contrast, the available processor cores are better utilized for parallelizing linear algebra operations if the feature sparsity is small.

5.3 Parallelized prediction

In addition to the use of multi-threading to speed up training, parallelization can also be used when predictions for several examples should be obtained. Delivering predictions for a given set of examples requires first enumerating the rules in a model and identifying those rules that cover each of the provided examples. Second, the predictions provided by the heads of these rules must be aggregated to obtain an overall prediction. As both of these steps may be carried out independently for each example, multi-threading can be used to predict for different examples in parallel. However, prediction time is usually not a limiting factor when dealing with datasets

comparable to those used in this work in terms of dimensionality. Hence, we leave it at the mention of this possibility and forego a closer examination of its advantages and disadvantages.

6 Conclusions, limitations and future work

We provided a detailed discussion of the pre-sorted search algorithm used for the efficient induction of rules by many successful rule learning approaches. We also discussed extensions to this algorithm that allow dealing with nominal attributes and missing feature values. Furthermore, we demonstrated how sparse data structures can be used to represent feature values. Our experiments suggest that the exploitation of feature sparsity drastically improves training efficiency in many cases. These empirical results are complemented by a study on the histogram-based induction of rules. We showed that the latter helps to reduce training times on datasets with low feature sparsity, where the ability to use sparse representations does not provide any benefits. Although the algorithmic principles discussed in this work are widely adopted by existing algorithms for the construction of tree- or rule-based models, publications on the topic are usually restricted to a high-level overview of the discussed techniques. The present work complements existing literature by providing an extensive and unified view of commonly used optimizations and approximations. Moreover, the empirical investigation using real-world multi-label datasets provided valuable insights for the practical use of rule learning algorithms. In particular, this applies to the idea of speeding up different aspects of a multi-label rule learning algorithm through parallelization. We observed that even though parallelization helps to reduce training times in many cases, its benefits heavily depend on the characteristics of a dataset and the learning algorithm at hand. The experimental results presented in this work helped us provide sane defaults for the parameters of the publicly available BOOMER algorithm that can be expected to work well in practice.

It should be noted, however, that empirical results regarding the runtime of algorithms do generally depend on low-level implementation details and the hardware used. Therefore, our experiments should not be considered as a replacement for benchmarks that must unavoidably be conducted when optimizing a particular implementation in a specific environment. Moreover, even though our experiments demonstrate the potential of the presented optimizations and approximations, they are not guaranteed to provide similar results when integrated into different learning algorithms.

Similar to the histogram-based approach examined in this work, unsupervised or supervised sampling techniques like *Gradient-based One-side Sampling* (Ke et al. 2017) may reduce the number of candidate rules to be evaluated by a rule learning algorithm and speed up training. However, compared to the histogram-based algorithm, we have found that sampling techniques of this kind have a more severe impact on predictive performance. Hence, we refrained from elaborating on this topic. A more promising direction for future work is motivated by the recent success of GPU-accelerated decision tree learners. Due to the similarities between decision trees and rule-based models, the underlying ideas used by such approaches (e.g., Mitchell and Frank 2017) can easily be generalized to rule learning algorithms.

Acknowledgments This work was partly funded by the German Research Foundation (DFG) under grant numbers 400845550 and 438445824.

Funding Open Access funding enabled and organized by Projekt DEAL.

Appendix A: Datasets

In Table 5, we provide an overview of the multi-label benchmark datasets that are used in this work. For a broad comparison of different algorithms, we include datasets of varying complexity that vastly differ in the number of examples, attributes and labels. All of these datasets can be downloaded from a publicly available repository at https://github.com/mrapp-ke/Boomer-Datasets.

Table 5 Characteristics of selected multi-label benchmark datasets from different domains, including the number of examples, numerical and nominal attributes and labels

Dataset	Domain	Examples	Attributes		Labels
			Numerical	Nominal	
20NG	Text	19, 300	1006	0	20
Bibtex	Text	7395	0	1836	159
Birds	Audio	645	258	2	19
Bookmarks	Text	87, 860	0	2150	208
Delicious	Text	16, 110	0	500	983
Emotions	Music	593	72	0	6
Enron	Text	1702	0	1001	53
EukaryoteGO	Biology	7766	12, 690	0	22
EukaryotePseAAC	Biology	7766	440	0	22
EUR-Lex-SM	Text	19350	5000	0	201
Image	Image	2000	294	0	5
IMDB	Text	120, 900	1001	0	28
Langlog	Text	1460	1004	0	75
Mediamill	Video	43, 910	120	0	101

Table 5 (continued)

Dataset	Domain	Examples	Attributes		Labels
			Numerical	Nominal	
Medical	Text	1954	1909	0	45
Nus-Wide cVLADplus	Image	269, 648	128	1	81
Ohsumed	Text	13, 930	1002	0	23
Reuters-K500	Text	6000	500	0	103
Scene	Image	2407	294	0	6
Slashdot	Text	3782	3125	0	22
TMC2007	Text	28, 600	0	49, 060	22
Yahoo-computers	Text	12, 444	34, 096	0	33
Yahoo-reference	Text	8027	39, 679	0	33
Yahoo-science	Text	6428	37, 187	0	40
Yahoo-social	Text	12, 111	52, 350	0	39
Yeast	Biology	2417	103	0	14

Appendix B: Predictive performance

In Sects. 3 and 5, we do not report the predictive performance achieved by the BOOMER algorithm, as they focus on computational efficiency and because the considered implementation variants do not affect algorithmic behavior. For completeness, an overview of the predictive performance achieved by the different configurations of the BOOMER algorithm used in Sects. 3 and 5 is given in Tables 6, 7 and 8. We assess predictive performance in terms of the Hamming loss, the subset 0/1 loss and the example-wise F1 measure. No parameter tuning was conducted. Instead, we used the algorithm's default parameters. Accordingly, for optimizing the Hamming loss, we learned 1, 000 single-label rules that minimize a decomposable variant of the logistic loss. For optimizing the subset 0/1 loss, we learned multi-label rules with respect to a non-decomposable variant of this loss function. A definition of these loss functions is provided by Rapp et al. (2020).

Table 6 Predictive performance achieved by different configurations of the BOOMER algorithm in terms of the Hamming loss (smaller values are better)

Dataset	Decomposable		Non-decomposable
	With feature Sampling	Without feature Sampling	
Hamming loss			
20NG	3.24	2.92	2.82
Bibtex	1.36	1.31	1.35
Birds	4.09	3.93	3.87
Bookmarks	0.91	0.91	–
Delicious	1.93	1.93	–
Emotions	18.47	19.22	18.61
Enron	4.60	4.61	4.74
EukaryoteGO	3.60	1.96	1.98
EukaryotePseAAC	5.05	5.12	5.79
EUR-Lex-SM	0.89	0.78	0.48
Image	15.06	14.80	14.18
IMDB	7.14	7.16	7.85
Langlog	1.52	1.54	1.52
Mediamill	3.16	–	–
Medical	1.30	0.88	0.87
Ohsumed	5.67	5.54	5.89
Reuters-K500	1.25	1.21	1.16
Scene	8.14	7.90	6.84
Slashdot	4.55	4.08	4.72
TMC2007	8.10	–	–
Yahoo-computers	3.47	3.15	3.57
Yahoo-reference	2.75	2.25	2.93
Yahoo-science	3.30	2.75	3.60
Yahoo-social	2.44	1.79	2.36
Yeast	20.05	19.95	19.24

Table 7 Predictive performance achieved by different configurations of the BOOMER algorithm in terms of the subset 0/1 loss (smaller values are better)

Dataset	Decomposable		Non-decomposable
	With Feature Sampling	Without Feature Sampling	
Subset 0/1 loss			
20NG	59.02	51.34	29.44
Bibtex	93.83	89.91	80.58
Birds	46.53	46.38	46.37
Bookmarks	88.76	87.87	–
Delicious	99.91	99.91	–
Emotions	69.81	70.99	66.79
Enron	90.72	89.96	84.02
EukaryoteGO	65.70	30.88	28.05
EukaryotePseAAC	86.21	86.47	66.86
EUR-Lex-SM	89.79	82.16	51.92
Image	55.70	54.00	43.40
IMDB	99.68	98.56	88.94
Langlog	81.44	80.00	80.27
Mediamill	93.01	–	–
Medical	47.04	29.27	25.13
Ohsumed	79.95	77.12	70.97
Reuters-K500	84.12	79.48	55.27
Scene	39.55	38.97	23.89
Slashdot	74.46	62.67	50.19
TMC2007	87.01	–	–
Yahoo-computers	63.98	58.31	54.41
Yahoo-reference	67.19	57.18	49.66
Yahoo-science	88.25	71.39	63.47
Yahoo-social	68.19	47.06	42.99
Yeast	86.10	85.40	76.95

Table 8 Predictive performance achieved by different configurations of the BOOMER algorithm in terms of the example-wise F1 measure (larger values are better)

Dataset	Decomposable		Non-decomposable
	With feature Sampling	Without feature Sampling	
Example-wise F1 measure			
20NG	43.11	51.06	72.72
Bibtex	14.85	24.54	41.23
Birds	62.18	64.10	63.71
Bookmarks	12.33	13.22	–
Delicious	0.93	0.92	–
Emotions	60.98	60.17	65.51
Enron	51.44	54.02	56.19
EukaryoteGO	39.05	80.53	81.63
EukaryotePseAAC	17.11	16.67	41.18
EUR-Lex-SM	24.80	38.10	69.55
Image	56.60	57.70	70.41
IMDB	0.70	3.87	27.79
Langlog	19.73	21.33	20.95
Mediamill	50.61	–	–
Medical	63.13	81.03	84.19
Ohsumed	35.81	42.01	51.73
Reuters-K500	18.46	23.92	56.51
Scene	65.29	66.13	80.92
Slashdot	29.35	44.66	57.88
TMC2007	45.35	–	–
Yahoo-computers	44.91	51.29	56.69
Yahoo-reference	36.36	47.41	56.82
Yahoo-science	14.38	35.69	44.71
Yahoo-social	35.48	59.19	63.85
Yeast	60.05	60.43	62.92

References

Alsabti K, Ranka S, Singh V (1998) CLOUDS: a decision tree classifier for large datasets. In: Proceeding international conference on knowledge discovery and data mining, p 2–8

Anderson E, Bai Z, Bischof C, et al (1999) LAPACK Users' guide. SIAM

Bénard C, Biau G, Da Veiga S et al (2021) SIRUS: Stable and interpretable RUle set for classification. Electronic J Stat 15(1):427–505

Blackford LS, Petitet A, Pozo R et al (2002) An updated set of basic linear algebra subprograms (BLAS). ACM Transact Math Softw 28(2):135–151

Boley M, Teshuva S, Bodic PL, et al (2021) Better short than greedy: interpretable models through optimal rule boosting. In: Proc. SIAM international conference on data mining, pp 351–359

Boström H (1995) Covering vs. divide-and-conquer for top-down induction of logic programs. In: Proc. international joint conference on artificial intelligence (IJCAI), pp 1194–1200
Breiman L (2001) Random forests. Mach Learn 45(1):5–32
Chen T, Guestrin C (2016) XGBoost: a scalable tree boosting system. In: Proc. ACM SIGKDD international conference on knowledge discovery and data mining, pp 785–794
Cohen WW (1995) Fast effective rule induction. In: Proc. International conference on machine learning (ICML), pp 115–123
Cohen WW, Singer Y (1999) A simple, fast, and effective rule learner. In: Proc. AAAI conference on artificial intelligence, pp 335–342
Conde D, Fernández MA, Rueda C et al (2021) Isotonic boosting classification rules. Adv Data Anal Classif 15:289–313
Dembczyński K, Kotłowski W, Słowiński R (2010) ENDER: a statistical framework for boosting decision rules. Data Min Knowl Disc 21(1):52–90
Dembczyński K, Waegeman W, Cheng W et al (2012) On label dependence and loss minimization in multi-label classification. Mach Learn 88(1–2):5–45
Du M, Liu N, Hu X (2019) Techniques for interpretable machine learning. Commun ACM 63(1):68–77
Friedman JH, Popescu BE (2008) Predictive learning via rule ensembles. Annals Appl Stat 2(3):916–954
Friedrich S, Antes G, Behr S et al (2022) Is there a role for statistics in artificial intelligence? Adv Data Anal Classif 16(4):823–846
Fürnkranz J (1999) Separate-and-conquer rule learning. Artif Intell Rev 13(1):3–54
Fürnkranz J (2005) From local to global patterns: evaluation issues in rule learning algorithms. In: Local pattern detection. Springer, p 20–38
Fürnkranz J, Gamberger D, Lavrač N (2012) Foundations of rule learning. Springer Science & Business Media
Gamberger D, Lavrač N (2000) Confirmation rule sets. In: Proc. European conference on principles of data mining and knowledge discovery (PKDD), pp 34–43
Gibaja E, Ventura S (2014) Multi-label learning: a review of the state of the art and ongoing research. Wiley Interdiscip Rev Data Mining Knowl Discov 4(6):411–444
Hall M, Frank E, Holmes G et al (2009) The WEKA data mining software: an update. SIGKDD Explor 11(1):10–18
Hüllermeier E, Fürnkranz J, Loza Mencía E, et al (2020) Rule-based multi-label classification: challenges and opportunities. In: International joint conference on rules and reasoning, pp 3–19
Jin R, Agrawal G (2003) Communication and memory efficient parallel decision tree construction. In: Proc. SIAM international conference on data mining, pp 119–129
Kamath C, Cantú-Paz E, Littau D (2002) Approximate splitting for ensembles of trees using histograms. In: Proc. SIAM international conference on data mining, pp 370–383
Ke G, Meng Q, Finley T et al (2017) LightGBM: a highly efficient gradient boosting decision tree. Adv Neural Inf Process Syst 30:3146–3154
Kotsiantis SB, Kanellopoulos D (2006) Discretization techniques: a recent survey. GESTS Int Transact Comput Sci Eng 32(1):47–58
Lakkaraju H, Bach SH, Leskovec J (2016) Interpretable decision sets: a joint framework for description and prediction. In: Proc. ACM SIGKDD international conference on knowledge discovery and data mining, pp 1675–1684
Langley P (1996) Elements of machine learning. Morgan Kaufmann
Li P, Wu Q, Burges C (2007) McRank: Learning to rank using multiple classification and gradient boosting. Adv Neural Inform Process Syst 20
Loza Mencía E, Fürnkranz J, Hüllermeier E, et al (2018) Learning interpretable rules for multi-label classification. In: Explainable and interpretable models in computer vision and machine learning. Springer, p 81–113
Mehta M, Agrawal R, Rissanen J (1996) SLIQ: a fast scalable classifier for data mining. In: Proc. International conference on extending database technology, pp 18–32
Mitchell R, Frank E (2017) Accelerating the XGBoost algorithm using GPU computing. PeerJ Comput Sci 3:e127
Mitchell TM (1997) Machine learning. McGraw Hill
Molnar C, Casalicchio G, Bischl B (2020) Interpretable machine learning – a brief history, state-of-the-art and challenges. In: Proc. European conference on machine learning and knowledge discovery in databases (ECML-PKDD), pp 417–431

Murdoch WJ, Singh C, Kumbier K et al (2019) Definitions, methods, and applications in interpretable machine learning. Proc Natl Acad Sci 116(44):22,071-22,080
Pagallo G, Haussler D (1990) Boolean feature discovery in empirical learning. Mach Learn 5(1):71–99
Rapp M (2021) BOOMER-an algorithm for learning gradient boosted multi-label classification rules. Softw Impacts 10(100):137
Rapp M, Loza Mencía E, Fürnkranz J, et al (2020) Learning gradient boosted multi-label classification rules. In: Proc. european conference on machine learning and knowledge discovery in databases (ECML-PKDD), pp 124–140
Rapp M, Loza Mencía E, Fürnkranz J, et al (2021) Gradient-based label binning in multi-label classification. In: Proc. european conference on machine learning and knowledge discovery in databases (ECML-PKDD), pp 462–477
Ribeiro MT, Singh S, Guestrin C (2016) "why should i trust you?" explaining the predictions of any classifier. In: Proc. ACM SIGKDD international conference on knowledge discovery and data mining, pp 1135–1144
Rivest RL (1987) Learning decision lists. Mach Learn 2(3):229–246
Shafer JC, Agrawal R, Mehta M (1996) SPRINT: a scalable parallel classifier for data mining. In: Proc. international conference on very large data bases, pp 544–555
Shi H (2007) Best-first decision tree learning. PhD thesis, University of Waikato
Singh C, Nasseri K, Tan YS et al (2021) imodels: a python package for fitting interpretable models. J Open Source Softw 6(61):3192
Vojíř S, Kliegr T (2020) Editable machine learning models? A rule-based framework for user studies of explainability. Adv Data Anal Classif 14(4):785–799
Weiss SM, Indurkhya N (2000) Lightweight rule induction. In: Proc. international conference on machine learning (ICML), pp 1135–1142
Wohlrab L, Fürnkranz J (2011) A review and comparison of strategies for handling missing values in separate-and-conquer rule learning. J Intell Inform Syst 36(1):73–98
Zilke JR, Loza Mencía E, Janssen F (2016) DeepRED – rule extraction from deep neural networks. In: Proc. international conference on discovery science, pp 457–473