Politecnico di Torino

Politecnico di Torino

Master Degree in Data Science and Engineering

Academic year 2023/2024

October 2024

MID: A New Strategy for Learning Optimal Decision Trees on Continuous Data

Supervisors

Prof. Elena Baralis

Prof. Siegfried Nijssen

Candidate

Antonio Dal Maso

ACKNOWLEDGEMENTS

Computational resources have been provided by the supercomputing facilities of the Université catholique de Louvain (CISM/UCL) and the Consortium des Équipements de Calcul Intensif en Fédération Wallonie Bruxelles (CÉCI) funded by the Fond de la Recherche Scientifique de Belgique (F.R.S.-FNRS) under convention 2.5020.11 and by the Walloon Region.

Contents

List of Figures

List of Tables

List of Algorithms

Introduction

The discretization of continuous features is a common preprocessing step in many machine learning applications, and, among data mining algorithms, decision tree learners (e.g., CART [1], C4.5 [2]) are closely related to discretization techniques: their output consists of models that discretize the input space in order to perform their task. Using discrete features rather than continuous ones can be beneficial in several ways: the work of Liu et al. [3] highlights how these are closer to a knowledge-level representation, making them more interpretable and easier to explain. Moreover, their usage can sometimes speed up the execution of data mining algorithms and significantly improve their performance [4]. In this work, another advantage of discretization is exploited: it allows techniques that normally cannot handle continuous attributes to be applied in a wider range of scenarios. An example belonging to this category is DL8.5 [5], an optimal decision tree learner (ODT) that only works on binary data. The trees produced by this algorithm are "optimal" because they achieve the smallest possible training error.

The problem of finding an optimal decision tree is NP-hard, even when constraints are specified to reduce the amount of possible solutions (i.e., on the depth of the tree). As a result, greedy algorithms have traditionally been favored over optimal ones due to their efficiency. However, recent advancements in optimization solvers and novel algorithmic concepts have led to the development of new optimal methods for inferring decision trees under constraints [6, 7, 8, 9, 10, 11], and empirical evidence demonstrates that these methods achieve improved classification performance on unseen data [12]. Since such approaches are often tailored to binary features, it is necessary to first preprocess data by discretizing continu-

ous features. Algorithms of this kind pose an additional challenge to discretization techniques: the choice of how binary features are produced heavily affects the performance of the models trained on them, and common heuristic-based discretizers give no guarantees about the optimality of their solutions. Moreover, binarizing a dataset can result in a considerable increase in its dimensionality, with consequences for the runtime of the algorithms that use it. For instance, one strategy that would allow an ODT algorithm to learn an optimal tree on a continuous dataset is to create a binary feature for every possible value of every feature. Doing so would be effective, but no optimal algorithm would complete its execution in a reasonable amount of time, unless in the case of a trivial starting dataset.

The goal of this thesis is to develop a discretization technique specifically designed for use with ODT algorithms, incorporating two key properties:

- It should generate a ranked list of binary features, allowing an ODT learner to be run iteratively with data of increasing dimensionality.
- If the process is stopped early, using only a subset of the binary features, the resulting models should still be of high quality.

A discretizer with these properties is desirable because it enables the learning of an optimal decision tree from continuous data in an anytime fashion: running this iterative process without a time limit leads to an optimal decision tree, but it can also be stopped early when waiting for it to terminate is unfeasible. This concept is implemented in MID (Minimum Impurity Discretizer), a new supervised, heuristic-based discretization technique that uses impurity measures such as the Gini index or entropy to produce its output. Experiments on well-known continuous datasets show that combining MID with optimal decision tree learning algorithms such as DL8.5 provides better results than classic discretization approaches and greedy methods like CART, even when small numbers of binary features are considered.

This thesis is organized as follows: the next chapter provides a general overview of decision trees and discretization techniques, with a detailed explanation of the algorithms directly involved in this work. Chapter 2 describes the Minimum Impurity Discretizer, and chapter 3 contains an analysis of the experiments performed to validate its efficacy, together with the related results.

Even though this analysis focuses on ODT learners (specifically on DL8.5), MID can be paired with any machine learning algorithm that works in a classification setting: a dataset must include class labels in order to be discretized using MID.

Chapter 1: Technical Backgroung

The subject of this work finds its roots into two widely studied topics of Machine Learning: decision trees and discretization techniques. The goal of this first chapter is to give to the reader a comprehensive description of the specific algorithms that were involved in the execution of this master thesis work, together with an overview of the basics of such subjects.

1.1 Decision Trees: an overview

Decision trees constitute a widely used category of machine learning models, which became popular because they are easy to interpret and very versatile: they can be used both in classification and regression domains, and they can also be part of more complex techniques (i.e. Random Forests [13], AdaBoost [14]).

The process used to assign a label to an instance of a dataset consists in applying a sequence of simple if-then-else decision rules on its features. Such rules are organized in a tree structure, where features are tested in the inner nodes and leaves correspond to class labels. Therefore, classifying an observation consists in following one of the paths of the tree connecting the root to a leaf. It is common for nodes testing categorical features to have a branch for every value that the feature can assume. Continuous attributes, on the other hand, are usually tested using thresholds, but they can also be made categorical before building the tree. In the simplest setting, binary trees are considered. Figure 1.1 shows the results of learning a decision tree on a synthetic dataset characterized by two continuous

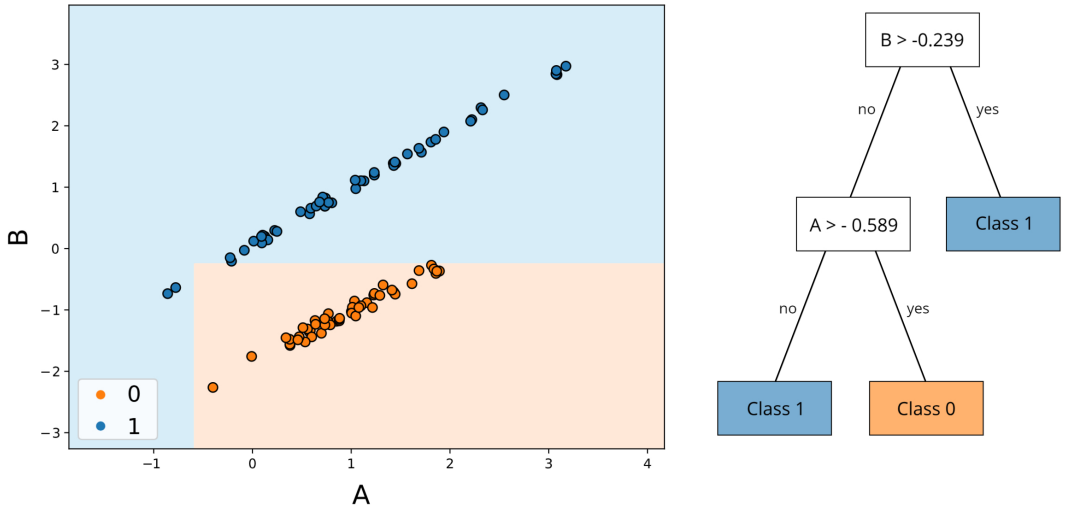


Figure 1.1: Left: a linearly separable synthetic dataset. Right: the decision tree trained on the synthetic dataset.

features. The model recursively partitions the training data into non-overlapping rectangular regions, such that samples corresponding to a specific region belong to the same class. It is possible to notice how the decision boundary of the classifier does not suit the dataset perfectly: the two classes are clearly linearly separable, and a single hyperplane would be enough to divide them. Yet, the decision tree needs two conditions to perform classification. This behaviour is one of the limits of univariate decision trees, namely trees that test a single feature for every node. Decision trees that test more than one feature at a time, usually by considering linear combinations of them, are referred to as multivariate.

Given a specific dataset, there exist a large number of different decision trees capable of addressing the same task. Exploring all of them to find the one achieving the highest performance is unfeasible in most scenarios. For this reason, the most popular decision tree learners are greedy. Such algorithms are highly scalable, easy to use, and can handle both categorical and continuous data without the need for preprocessing. They build trees by locally selecting features at each node using a heuristic, starting from the root and progressing down to the leaf

nodes. Once this process is completed (e.g. when all the leaves are pure), the tree might be too complex for the problem at hand, and it may not generalize well to unseen data [15]. For this reason, the tree might undergo a pruning process to reduce overfitting issues. Alternatively, a set of constraints can be defined as stopping conditions. For instance, the tree could be grown until further splitting a node would result in leaves with too few samples. In the case of univariate decision trees, it is common to use an impurity-based criterion to decide which attribute should be tested in each node. Some algorithms in this category include CART [1], C4.5 [2], and ID3 [16].

Even though finding an Optimal Decision Tree (ODT) is NP-hard, it is still interesting and worthwhile to attempt solving such a problem. Optimal algorithms indeed present several advantages [7]:

- Consider the case where the learner must find a tree within certain constraints. This is a common scenario, as constraints reduce the dimensionality of the search space and make the problem more manageable. If an optimal algorithm completes the training process and does not return a solution, it indicates that the problem cannot be solved within the given constraints. Conversely, with a greedy learner, the chosen strategy might not be sufficient to find a solution, even if one exists.

- Optimal algorithms can be used as a benchmark to evaluate the performance and quality of greedy techniques.

In recent years, the development of optimization solvers and innovative algorithmic concepts has led to new methods for inferring optimal decision trees. These methods include mixed integer programming [6, 11, 12, 17], constraint programming [10], SAT solvers [18], and dynamic programming [5, 7, 8]. Among these, dynamic programming techniques are noted for their speed and accuracy, especially under depth constraints. The focus of this work is one such technique:

DL8.5 [5]. This algorithm is discussed in detail in Section 1.3, after a comprehensive overview of CART (this serves as a comparison for evaluating the performance of DL8.5 when paired with the Minimum Impurity Discretizer).

1.2 CART (Classification and Regression Trees)

Algorithm 1 presents the pseudocode of CART, a greedy algorithm that can handle both continuous and discrete features. The implementation provided below is designed to work only with numerical attributes, but it can be easily adapted to process categorical data as well. CART recursively partitions the training data into smaller subsets using binary splits, starting at the root node and continuing until a stopping criterion is met. At each node, it selects the feature and threshold that best separate the data, optimizing a metric which represents how homogeneous the resulting subsets are. This version of the algorithm involves three stopping criteria: a maximum tree depth, a minimum number of instances per leaf node, and a purity check for the leaves.

Algorithm 1 CART

Require: $S = \{(\mathbf{x_1}, y_1), \ldots, (\mathbf{x_n}, y_n)\} \longrightarrow$ *labeled training dataset*
Require: $D \longrightarrow$ *current tree depth*
Require: $n_0 \longrightarrow$ *min amount of instances per leaf node*
Require: $D_{max} \longrightarrow$ *maximum tree depth*
Create a tree T with a single root node
if S is pure **or** $|S| < n_0$ **or** $D \geq D_{max}$ **then**
    Assign majority class label to root of T and mark it as leaf node
else
    $min\_imp \leftarrow +\infty$
    for feature f **do**
        $t, imp \leftarrow$ `best_split_single_feature`(S, f)
        if $imp < min\_imp$ **then**
            $min\_imp \leftarrow imp$
            $f_b \leftarrow f$
            $t_b \leftarrow t$
    $S_{left} \leftarrow \{(\mathbf{x}, y) \in S : \mathbf{x}(f_b) \leq t_b\}$
    $S_{right} \leftarrow \{(\mathbf{x}, y) \in S : \mathbf{x}(f_b) > t_b\}$
    Connect output of `CART`$(S_{left}, D+1, n_0, D_{max})$ to T as left child
    Connect output of `CART`$(S_{right}, D+1, n_0, D_{max})$ to T as right child
return T

The quantity t provided by `best_split_single_feature(`S, f`)` is the threshold that should be used to test feature f at the node of the tree associated with the set of observations S. When CART is used for classification tasks, t is typically computed by finding the value that minimizes the Gini index (definition 1). [15] In this context, the value *imp* returned along with t corresponds to $Gini(t; S)$. This represents the minimum impurity level achievable by testing feature f, and it can be used to identify the best feature for separating the data.

Definition 1 (Gini index)**.** *Let $S = \{a_1, \ldots, a_N\}$ be a set of N observations associated to a continuous feature f, each of which is characterized by a class c_i (k different classes are available). The Gini index associated to subset S is given by the relation*

$$\begin{aligned} Gini(S) &= \sum_{i=1}^{k} P(c_i|S)(1 - P(c_i|S)) \\ &= 1 - \sum_{i=1}^{k} P(c_i|S)^2 \ , \end{aligned}$$

where $P(c_i|S)$ is the proportion of examples in S belonging to class c_i. Let now T be a candidate split, partitioning S into two subsets $S_1 = \{a \in S : a \leq T\}$ and $S_2 = \{a \in S : a > T\}$; the Gini index associated to T is defined as

$$Gini(T; S) = \frac{|S_1|}{|S|} Gini(S_1) + \frac{|S_2|}{|S|} Gini(S_2) \ .$$

The more homogeneous the classes within S_1 and S_2, the smaller the $Gini(T; S)$.

This procedure is analogous to the approach used in many greedy discretization techniques. In fact, CART can be used on continuous datasets without the need for preprocessing because it applies its own form of discretization on the data. It is important to note that several versions of CART exist, differing in aspects such as the stopping criteria and the metrics used to measure node homogeneity. However,

the core structure of the algorithm remains consistent. This section describes the characteristics of the specific version used in this work for comparison with MID and DL8.5.

1.3 DL8.5

DL8.5 is an ODT learner that relies on itemset mining techniques, and it can only work on binary data. These are first converted into a transactional database: the observations composing the dataset are represented as sets of items, where each item encodes the value of one original binary feature. The set of items representing an observation is referred to as "transaction". Formally, a transactional database is a collection $\mathcal{D} \coloneqq \{(t, I, c) \mid t \in \mathcal{T}, I \subseteq \mathcal{I}, c \in \mathcal{C}\}$, where $\mathcal{T}$ is a set of transaction identifiers, $\mathcal{I}$ is the set of possible items, and $\mathcal{C}$ is the set of class labels. $\mathcal{I}$ contains two items for each original binary feature, one encoding the value 1 and one encoding the value 0. Each transaction can only contain one of these two items for a single feature. Table 1.1 contains an example binary dataset, together with the transactional database that corresponds to it. The observations are characterized by three binary features (A, B and C). Thus, the set of possible items is $\mathcal{I} = \{a, \neg a, b, \neg b, c, \neg c\}$.

A key result of using this representation is that any path of a decision tree learned on the dataset can be described using an itemset I. The lattice reported

A	**B**	**C**	**class**
1	*1*	*1*	*1*
1	*1*	*0*	*1*
1	*0*	*1*	*0*
1	*0*	*0*	*1*
0	*1*	*1*	*0*
0	*1*	*0*	*0*
0	*0*	*1*	*0*
0	*0*	*0*	*0*

(a) Binary dataset

t	**transaction**	**class**
1	$a,\ b,\ c$	*1*
2	$a,\ b,\ \neg c$	*1*
3	$a,\ \neg b,\ c$	*0*
4	$a,\ \neg b,\ \neg c$	*1*
5	$\neg a,\ b,\ c$	*0*
6	$\neg a,\ b,\ \neg c$	*0*
7	$\neg a,\ \neg b,\ c$	*0*
8	$\neg a,\ \neg b,\ \neg c$	*0*

(b) Transactional database

Table 1.1: Example conversion of a binary dataset into a transactional database.

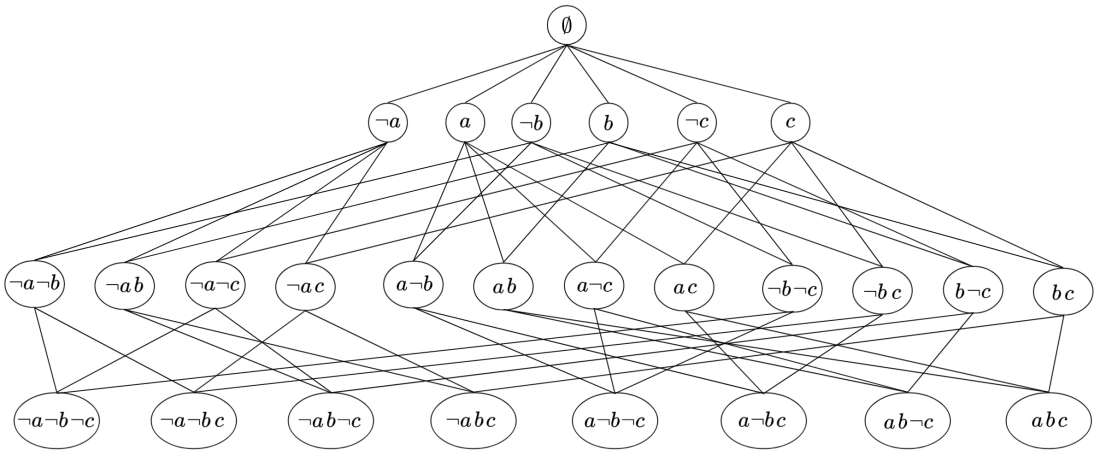


Figure 1.2: Complete search space of itemsets for the dataset of Table 1.1.

in Figure 1.2 illustrates all the possible itemsets for the toy dataset of table 1.1. In this context, a decision tree can be represented as a subset of the itemsets belonging to the lattice, and the branches of the decision tree correspond to subset relations. Let us define:

- The cover of an itemset I as the set of transactions containing I: $cover(I) = \{(t, X, c) | (t, X, c) \in \mathcal{D} \text{ and } I \subseteq X\}$.
- The class-based support of an itemset I as the number of examples in $cover(I)$ characterized by a given class c: $Sup(I, c) = |\{(t, X, \hat{c}) \in cover(I) \text{ and } \hat{c} = c\}|$.
- The error of an itemset I as the difference between the cardinality of $cover(I)$ and its maximum class-based support:

$$leaf\_error(I) = |cover(I)| - \max_{c \in \mathcal{C}} \{Sup(I, c)\} \ .$$

DL8.5 leverages a recursive depth-first approach to explore the space of the solutions (namely, the itemsets lattice). To be specific, it finds an optimal decision tree by looking for a collection of itemsets $\mathcal{DT}$ that minimizes the quantity

$$\sum_{I \in \mathcal{DT}} leaf\_error(I).$$

Algorithm 2 presents the pseudocode of DL8.5, whose core is `DL8.5-Recurse`. This recursive function returns an optimal decision tree for the transactions covered by an itemset I. The algorithm attempts to extend I using each binary feature, and selects the one yielding the smallest error. This process continues until the maximum depth constraint *maxdepth* is reached, the support of I falls below a threshold *minsup*, or I represents a pure leaf ($leaf\_error(I) = 0$). Additionally, the algorithm is anytime: it is possible to specify a time limit for the training process, after which the best solution found so far is returned.

A key characteristic of DL8.5 is its use of a branch-and-bound approach to prune parts of the solutions search space. The recursive function keeps track of the error associated with the best running solution, and uses this value as an upper bound to avoid unnecessary computations. Take line 18 as an example: during the evaluation of a binary feature, if the first branch of the tree already yields an error greater than the upper bound, the recursion on the second branch is avoided. The upper bound is initialized at $+\infty$, but it is updated as soon as a solution is found. Eventually, DL8.5 avoids redundant computations by storing the results of the calls to `DL8.5-Recurse` in a cache. This is effective because the same itemset can be reached through multiple paths in the lattice. A complete and more detailed analysis of the algorithm and its characteristics can be found in the original paper [5].

Algorithm 2 DL8.5

Require: $maxdepth$, $minsup$
1: **struct** $BestTree\{init\_ub : float; tree : Tree; error : float\}$
2: $cache \leftarrow HashSet < Itemset, BestTree >$
3: $bestSolution \leftarrow$ `DL8.5-Recurse`$(\emptyset, +\infty)$
4: **return** $bestSolution.tree$
5:
6: **Procedure** `DL8.5-Recurse`$(I, init\_ub)$
7: **if** $leaf\_error(I) = 0$ **or** $|I| = maxdepth$ **or** time-out is reached **then**
8: **return** $BestTree(init\_ub,$ `make_leaf`$(I), leaf\_error(I))$
9: $sol \leftarrow cache.get(I)$
10: **if** sol was found **and** $((sol.tree \neq$ `NO_TREE`$)$ **or** $(init\_ub \leq sol.init\_ub))$ **then**
11: **return** sol
12: $(\tau, b, ub) \leftarrow$ (`NO_TREE`$, +\infty, init\_ub)$
13: **for** all attributes i **do**
14: **if** $|cover(I \cup \{i\})| \geq minsup$ **and** $|cover(I \cup \{\neg i\})| \geq minsup$ **then**
15: $sol_1 \leftarrow$ `DL8.5-Recurse`$(I \cup \{\neg i\}, ub)$
16: **if** $sol_1.tree =$ `NO_TREE` **then**
17: **continue**
18: **if** $sol_1.error \leq ub$ **then**
19: $sol_2 \leftarrow$ `DL8.5-Recurse`$(I \cup \{i\}, ub - sol_1.error)$
20: **if** $sol_2.tree =$ `NO_TREE` **then**
21: **continue**
22: $feature\_error \leftarrow sol_1.error + sol_2.error$
23: **if** $feature\_error \leq ub$ **then**
24: $\tau \leftarrow$ `make_tree`$(i, sol_1.tree, sol_2.tree)$
25: $b \leftarrow feature\_error$
26: $ub \leftarrow b - 1$
27: **if** $feature\_error = 0$ **then**
28: **break**
29: $sol \leftarrow BestTree(init\_ub, \tau, b)$
30: $cache.store(I, sol)$
31: **return** sol

1.4 Discretization Techniques: an overview

Datasets usually contain three types of attribute: categorical, discrete or continuous. Both discrete and continuous attributes are numerical, but while the former ones are countable, the latter ones are not. Finally, the values assumed by categorical variables are not characterized by an order. Discretization is the process of turning a continuous attribute into a discrete one, and it is done by dividing its full range of values into a set of intervals that is often finite and small. It is characterized by two key steps:

- choosing the appropriate number of intervals;
- finding the most suitable thresholds to separate the values.

It is common for discretization algorithms to need for the user to specify the number of intervals in advance. For example, two of the earlier and best-know strategies of this kind consist in dividing the range of an attribute into sub-ranges either having an equal width or containing the same number of instances (these approaches are referred to as "equal-width" and "equal-frequency", respectively). The need for accurate and efficient classification has been growing during the past years, and many discretization algorithms (more complex and effective than the two described above) have been proposed and tested to satisfy such demand. Generally speaking, previous works regarding these methods [3, 19, 20, 21, 22] categorize them as:

- ***Supervised vs. Unsupervised***. Discretizers are supervised if they use class information, and unsupervised if they do not. Even though unsupervised methods are the only choice when class labels are not available, they present some limitations. As an example, equal-width and equal-frequency may not give good results if the distribution of the attribute is not uniform, and they are very sensitive to outliers [23]. Some comparative studies suggest that supervised methods achieve better performance than unsupervised ones [4, 22, 24], but this is actually very related to the nature and the size of the dataset. Both strategies can be the best one depending on the specific scenario [20, 25].

- ***Dynamic vs. Static***. A discretizer is defined as dynamic if it is embedded into a data mining algorithm (i.e. the decision tree learners C4.5 [2] and CART [1]), while it is considered static if it is applied to the data as an independent process. Most of the known discretization techniques fall into the second category.

- ***Global vs. Local***. This distinction depends on the amount of information exploited by the algorithm to operate on an attribute: a discretizer is global if it requires all the available data to make a decision, while it is local if only a subset of it is needed. The dynamic discretizers embedded inside the decision tree learners are a perfect example of local algorithms, because they split the range of an attribute by looking only at the observations related to the node which is taken into consideration.

- ***Splitting vs. Merging***. This is related to the approach used to obtain new intervals as the discretization progresses. In splitting (or top-down) methods, a single interval is divided into two (or more) smaller ones by identifying at every iteration the best threshold among the available ones. On the other hand, merging (or bottom-up) discretizers start from a predefined set of intervals (i.e. one for every observation in the dataset) and iteratively combine them to get to the final result. This is not a hard classification, as some algorithms can alternate the splitting and merging actions during their execution [26, 27].

- ***Direct vs. Incremental***. Direct methods divide the range of an attribute into k intervals simultaneously, while incremental methods begin with a simple discretization and pass through an improvement process. When dealing with the former ones, the number k of intervals to be found must be known a priori; with incremental methods, instead, an additional criterion is needed to decide when to stop the discretization (e.g. a minimum number of observations for each interval, or a threshold on the purity of the resulting partition).

- ***Univariate vs. Multivariate***. Eventually, discretizers are distinguished depending on whether they operate on one feature at a time (univariate) or on multiple features simultaneously (multivariate). Notice that the meth-

ods falling in the latter category are inevitably characterized by a higher time complexity, and that they can overall be considered a more difficult challenge. In fact, in this case correlation and interaction effects among attributes must be properly taken into account.

According to Liu et al. [3], the univariate discretization process for a single attribute follows the steps illustrated in Figure 1.3. First, (1) the observations of the attribute are sorted in either increasing or decreasing order. Next, (2) an evaluation metric is applied to compare the available candidates and select the best one for a discretization step. A candidate can be either a cut-point to split an interval or a pair of intervals to be merged. After selecting the best candidate, (3) the splitting (or merging) operation is performed. These steps are repeated until a termination condition is met (4).

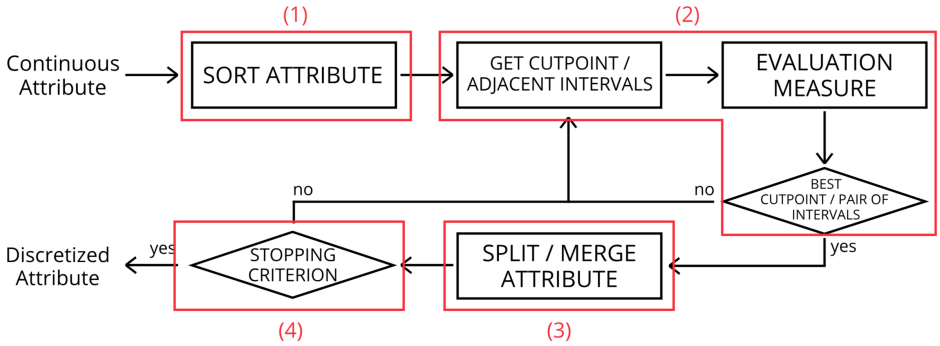


Figure 1.3: Univariate discretization process of a single continuous attribute.

The evaluation measure, the stopping criterion, and the splitting or merging strategy are what distinguish one discretization technique from another. For example, consider equal-width discretization: after the sorting phase, all cut-points are directly computed from the minimum and maximum values of the feature. The evaluation measure and stopping criterion are straightforward, as all computed thresholds are applied to split the attribute's range. Some commonly used discretizers are the ChiMerge algorithm [28] (a bottom-up method using the χ^2

value to determine the merging point) and the MDLP algorithm [29] (which follows an entropy-based top-down strategy). The latter one is analyzed in detail in the next section.

1.5 MDLP discretizer

The algorithm that we are about to investigate uses the Shannon entropy [23] to evaluate how good a candidate cut-point is. Let us start by giving a formal definition of this quantity.

Definition 2 (Shannon Entropy)**.** *Let X be a discrete random variable which takes values in an alphabet $\mathcal{X}$ and is distributed according to $p : \mathcal{X} \longrightarrow [0, 1]$. p is such that $p(x) := \mathbb{P}[X = x]$, $\forall x \in \mathcal{X}$. Then, the entropy of X is given by the relation*

$$H(X) = \sum_{x \in \mathcal{X}} p(x) \log_b \frac{1}{p(x)} = -\sum_{x \in \mathcal{X}} p(x) \log_b p(x) \, .$$

If the logarithm is in base 2 ($b = 2$), $H(X)$ is said to be measured in bits.

It is common to interpret the Shannon entropy as a measure of how uncertain the outcome of a random variable is: $H(X)$ is minimal if one of the x's always occurs (meaning that $\exists! x : p(x) = 1$), while it is maximal when all the values in $\mathcal{X}$ are equally likely. In the context of the discretization of a continuous attribute, this metric is used to evaluate the *pureness* of the intervals that would be produced from the application of a given cut-point.

Let us assume $S = \{a_1, \ldots, a_N\}$ to be a set of N observations associated to a continuous feature A, each of which is characterized by a class c_i (k different classes are available). Assuming that the classes are observations of a random variable C, an estimate of $H(C)$ can be computed as

$$H(C|S) = -\sum_{i=1}^{k} P(c_i|S) \log_2 P(c_i|S) \, ,$$

where $P(c_i|S)$ is the proportion of examples in S belonging to class c_i. $H(C|S)$ is the *class entropy* associated with subset S. Let T be a potential cut-point, partitioning S into two subsets $S_1 = \{a \in S : a \leq T\}$ and $S_2 = \{a \in S : a > T\}$. The class entropy associated to the partition induced by T is defined as

$$E(A, T; S) = \frac{|S_1|}{|S|} H(C|S_1) + \frac{|S_2|}{|S|} H(C|S_2) \; . \tag{1.1}$$

The more homogeneous the classes within S_1 and S_2, the smaller $E(A, T; S)$. The trivial scenario takes place when the two subsets are perfectly pure: in this case $H(C|S_1)$, $H(C|S_2)$ and $E(A, T; S)$ are all equal to 0. MDLP considers the cut-point T corresponding to the smallest class entropy as the best candidate to split the samples in S.

Intuitively, it is reasonable to use the entropy minimization heuristic in the context of discretization: what matters the most in many applications is the relationship between the features in the dataset and the target variable, thus, if a set of values are associated to a specific class, it makes sense to group them together. Moreover, the usage of this technique is also justified formally by theorem 1. Let us introduce the concept of *boundary point*.

Definition 3 (Boundary point)**.** *A value T in the range of A is a boundary point iff in the sequence of examples sorted by the value of A there exist two examples $e_1, e_2 \in S$, having different classes, such that $A(e_1) < T < A(e_2)$. Moreover, there exists no other example $e' \in S$ such that $A(e_1) < A(e') < A(e_2)$.*

Theorem 1. *If T minimizes the measure $E(A, T; S)$, then T is a boundary point.*

Proof. Let S contain three subsets P, Q and R such that all the samples in Q belong to class c_q, and

$$p = \sum_{j=1}^{k} p_j \; , \quad r = \sum_{j=1}^{k} r_j \; .$$

p_j and r_j are the numbers of instances of class j in P and R, respectively. Let us

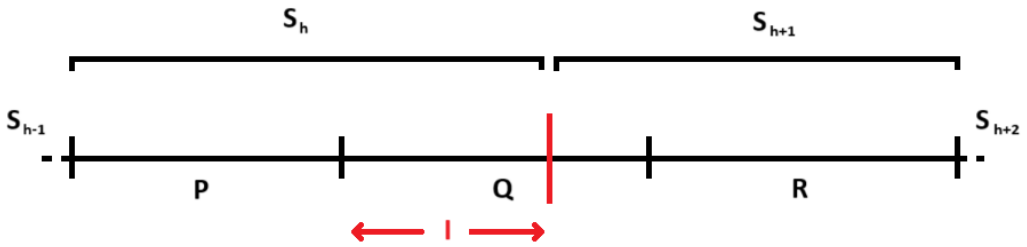


Figure 1.4: Scenario considered in the proof of theorem 1: S_h and S_{h+1} constitute a partition of the set $P \cup Q \cup R$.

consider the m-ary partition $S_1, \ldots, S_m$ of S, where S_h and S_{h+1} consist of the set $P \cup Q \cup R$ (Figure 1.4). Finally, let us assume to evaluate a candidate cut-point T such that l examples of Q belong to S_h and $|Q| - l$ to S_{h+1} (with $0 \leq l \leq |Q|$). Then, the class entropy associated with T can be written as

$$E(A, T; S) = \frac{L(l) + R(l)}{|S|} ,$$

$$\text{with} \quad L(l) = \sum_{i=1}^{h} |S_i| H(C|S_i) , \quad R(l) = \sum_{i=h+1}^{m} |S_i| H(C|S_i) .$$

Now,

$$\begin{aligned}
L(l) &= \sum_{i=1}^{h-1} |S_i| H(C|S_i) + |S_h| H(C|S_h) \\
&= \sum_{i=1}^{h-1} |S_i| H(C|S_i) + \sum_{j \in \{1,\ldots,k\} \setminus \{q\}} p_j \log \frac{p+l}{p_j} - (p_q + l) \log \frac{p_q + l}{p+l} \\
&= \sum_{i=1}^{h-1} |S_i| H(C|S_i) + \log(p+l) \cdot \sum_{j \in \{1,\ldots,k\} \setminus \{q\}} p_j + \log(p+l) \cdot (p_q + l) \\
&\quad - \sum_{j \in \{1,\ldots,k\} \setminus \{q\}} p_j \log p_j - (p_q + l) \log(p_q + l) \\
&= \sum_{i=1}^{h-1} |S_i| H(C|S_i) + (p+l) \log(p+l) - (p_q + l) \log(p_q + l) \\
&\quad - \sum_{j \in \{1,\ldots,k\} \setminus \{q\}} p_j \log p_j .
\end{aligned}$$

The two summations do not depend on l, thus

$$\begin{aligned} L''(l) &= \frac{\mathrm{d}}{\mathrm{d}l}\left[\log(p+l) - \log(p_q+l)\right] \\ &= \frac{1}{p+l} - \frac{1}{p_q+l} \\ &= \frac{p_q - p}{(p+l)(p_q+l)} \leq 0 \ . \end{aligned}$$

Similarly,

$$\begin{aligned} R(l) = \sum_{i=h+2}^{m} |S_i| H(C|S_i) &+ (r + |Q| - l)\log(r + |Q| - l) \\ &- (r_q + |Q| - l)\log(r_q + |Q| - l) - \sum_{j \in \{1,\ldots,k\}\setminus\{q\}} r_j \log r_j \end{aligned}$$

$$\implies R''(l) = \frac{r_q - r}{(r + |Q| - l)(r_q + |Q| - l)} \leq 0 \ .$$

This implies that $E''(A, T; S)$ is non-positive $\forall l$ as well, and that the minima of the class entropy associated to T can not belong to Q[1]. □

Theorem 1 implies that the heuristic we are considering is "well-behaved", meaning that it will never select a cut-point that separates consecutive examples belonging to the same class. Moreover, this theorem implies that only boundary points should be considered as candidates to become cut-points, thus reducing considerably the search space.

What characterizes the MDLP discretizer the most is the criterion used to decide when to stop its execution. The main contribution of Fayyad and Irani [29] was in fact the formulation of a *Minimum Description Length Principle* suitable for this task, which can be used to decide whether the cut-point associated with the minimum class entropy should actually be applied or not. This relies on the idea that the best alternative between two options is the one allowing the shortest

[1]Even though the natural logarithm has been used to make computations easier, the same result can be reached using the logarithm in base 2

description of the result. Despite of the relevance of such principle, it will not be further analyzed as it is not related to this master thesis work. A complete description can be found in the original paper. Algorithm 3 contains the pseudocode of the MDLP discretizer, which operates on a single continuous feature at a time.

Algorithm 3 MDLP discretizer

Require: $X = \{x_1, \ldots, x_n\} \longrightarrow$ *set of observations of a continuous feature* A
Require: $y \longrightarrow$ *set of class labels*
Compute *boundary_points* from X and y
cut_points $\leftarrow$ empty list [ ]
`MDLP-Recurse(`$X, y,$ *boundary_points*, *cut_points*`)`
return *cut_points*

Procedure `MDLP-Recurse(`$X, y,$ *boundary_points*, *cut_points*`)`
    $N\_unique \leftarrow$ n. of unique elements in X
    if $N\_unique < 2$ **then**
        return
    cut_point $\leftarrow$ `retrieve_best_cut_point(`$X, y,$ *boundary_points*`)`
    if no cut-point has been selected **then**
        return
    decision $\leftarrow$ `MDLP_criterion(`$X, y,$ *cut_point*`)`
    if $decision == False$ **then**
        return
    $X\_left \leftarrow \{x \in X : x \leq cut\_point\}$
    $X\_right \leftarrow \{x \in X : x > cut\_point\}$
    $y\_left \leftarrow$ labels corresponding to the values in X_left
    $y\_right \leftarrow$ labels corresponding to the values in X_right

    append *cut_point* to *cut_points*
    `MDLP-Recurse(`$x\_left, y\_left,$ *boundary_points*, *cut_points*`)`
    `MDLP-Recurse(`$x\_right, y\_right,$ *boundary_points*, *cut_points*`)`

`MDLP-Recurse` is a recursive procedure very similar to the depth-first visit of a tree. Two auxiliary functions are used: `MDLP_criterion` (which applies the minimum description length principle to accept or reject a cut-point), and `retrieve_best_cut_point` (that returns the boundary point minimizing the class entropy of X). The execution terminates when all the observations in X have the same value, or when the MDLP criterion rejects the cut.

1.6 Additional considerations about common discretizers

The final goal of this work, as outlined in the introduction, is to develop a discretizer compatible with an anytime approach for learning optimal decision trees on continuous data. The idea is to run an ODT algorithm iteratively, increasing the dimensionality of the binarized dataset with each iteration. But why is a new discretization technique necessary? Could existing ones be used to achieve this?

The discretizers introduced in earlier sections share a common limitation: they operate on each continuous feature individually and independently of the others. This poses challenges for the proposed anytime strategy: how should the range of each continuous feature be divided to generate exactly N binary features? What strategy should be used to add new features?

Furthermore, a desirable property would be that, at each iteration, the new set of binary features used for training is a superset of those used previously. This ensures that the ODT algorithm only finds a new solution if the newly added features provide additional value, resulting in a steadily increasing training accuracy. As a result, when the anytime process is stopped, the final solution is optimal with respect to all the solutions found up to that point.

Existing discretization techniques are not designed to meet these requirements, making them unsuitable for supporting this type of anytime approach. For this reason, the Minimum Impurity Discretizer was developed.

Chapter 2: Minimum Impurity Discretizer

This chapter introduces MID (Minimum Impurity Discretizer), a new supervised, heuristic-based, multivariate discretization technique. Its goal is to produce an ordered list of binary features from continuous input data. Binary datasets of different dimensionalities can then be generated by selecting shorter or longer prefixes of this list. This algorithm is related to the MDLP discretizer introduced in section 1.5: it uses a top-down strategy, where an impurity metric is used to evaluate the quality of a split. Moreover, it takes advantage of theorem 1 to reduce the search space of the candidate thresholds and speed up execution. Unlike MDLP, however, MID discretizes continuous features jointly, comparing the quality of thresholds across multiple features to find the best combination according to the heuristic.

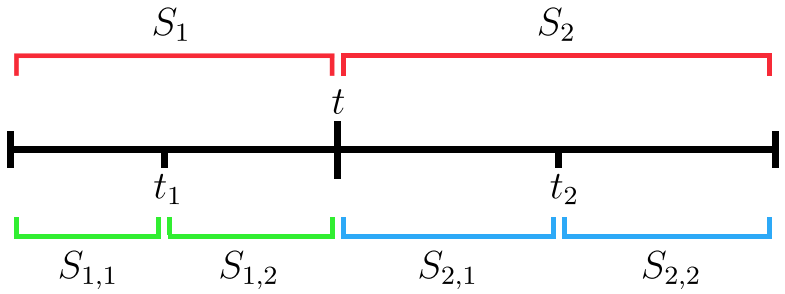


Figure 2.1: Sample discretization scenario: a threshold t divides the observations of a continuous feature into two sets S_1 and S_2. Two additional candidate thresholds t_1 and t_2 can be considered to further discretize the feature.

Assuming entropy is used as the impurity metric, MID applies the following preliminary operations to each continuous feature A:

1. The observations of A are reordered along with their class labels.

2. The boundary points are retrieved to be used as candidates. We use T_c to refer to the set containing all of them.

3. The class entropy related to A is computed for every t in T_c.

4. The candidate t^* with the smallest entropy value is stored separately, together with its entropy gain. This is the difference between the entropy of A before and after applying t^*.

5. t^* is removed from T_c.

6. Steps 3 to 5 are repeated until T_c is empty.

For example, consider the scenario in Figure 2.1, where $T_c = \{t_1, t_2\}$ and t (which was previously selected) divides the observations of A in two subsets S_1 and S_2. If applied, t_1 would split S_1 in $S_{1,1}$ and $S_{1,2}$. In the same way, t_2 would divide S_2 in $S_{2,1}$ and $S_{2,2}$. Let S be the full set of samples. The entropy scores associated to the candidates are

$$E(t_1) = \frac{|S_{1,1}|}{|S|} H(C|S_{1,1}) + \frac{|S_{1,2}|}{|S|} H(C|S_{1,2}) + \frac{|S_2|}{|S|} H(C|S_2) \ ,$$

$$E(t_2) = \frac{|S_1|}{|S|} H(C|S_1) + \frac{|S_{2,1}|}{|S|} H(C|S_{2,1}) + \frac{|S_{2,2}|}{|S|} H(C|S_{2,2}) \ .$$

Performing these operations for all the boundary points of a continuous feature establishes a ranking among them. Each element of the ranking is considered to be the best threshold to discretize the continuous feature, given that all the elements occupying a higher position have already been applied.

MID avoids redundant computation of class entropies by using a cache to store the terms $H(C|S_k)$ for every subset S_k. Let E_i denote the total entropy of the class labels at the end of the i-th iteration, after i thresholds have already been removed from T_c. Let t be a candidate that would split a subset S_k into $S_{k,1}$ and

$S_{k,2}$. Then, the entropy score associated with t at iteration $i+1$ is

$$E(t) = E_i - \frac{|S_k|}{|S|}H(C|S_k) + \frac{|S_{k,1}|}{|S|}H(C|S_{k,1}) + \frac{|S_{k,2}|}{|S|}H(C|S_{k,2}) \ .$$

By storing the terms $H(C|S_k)$, $H(C|S_{k,1})$ and $H(C|S_{k,2})$ at each iteration, it becomes easier to compute the class entropy for each candidate by leveraging previous computations.

Algorithm 4 contains the pseudocode for training a minimum impurity discretizer when applied to a full dataset. Lines 1 to 6 implement the steps described above: for each feature, boundary points are computed (line 4) and sorted using an impurity metric (line 5). A list containing the ranked boundary points and their associated gains is created and saved. It is then possible to produce a global ranking of binary features by merging together these lists (line 7).

Algorithm 4 MID training process

Require: Dataset X, target variable y
Require: $N \longrightarrow$ *number of desired binary features*
1: *sorted_b_points_all_features* ← empty list [ ]
2: **for** x **in** columns of X **do**
3: x and y are sorted based on the values of x
4: *b_points* ← `compute_boundary_points`(x, y)
5: *sorted_b_points* ← `sort_boundary_points`$(x, y,$ *b_points*$)$
6: *sorted_b_points* is appended to *sorted_b_points_all_features*
7: *thresholds* ← `get_best_thresholds`(*sorted_b_points_per_feature*, N)
8:
9: **Procedure** `compute_boundary_points`(x_{sorted}, y_{sorted})
10: *b_points* ← empty list [ ]
11: **for** x **in** x_{sorted} (starting from 2nd element) **do**
12: $x_p \leftarrow$ element preceding x
13: **if** $x_p \neq x$ **then**
14: *b_point* ← $(x_p + x)/2$
15: $y, y_p \leftarrow$ class label of x, x_p
16: **if** $y_p \neq y$ **then**
17: append *b_point* to *b_points*
18: **else**
19: *y_prev* ← set of labels of the samples equal to x_p
20: *y_next* ← set of labels of the samples equal to x
21: *y_merged* ← *y_prev* ∪ *y_next*
22: **if** `length`(*y_merged*) > 1 **then** append *b_point* to *b_points*
23: **return** *b_points*

```
Procedure sort_boundary_points(x_sorted, y_sorted, b_points)
    Initialize interval_cache, parent_cache
    sorted_b_points ← empty list [ ]
    E_{i-1} ← compute_entropy(y_sorted)
    for x in b_points do parent_cache.store(x, E_{i-1})
    N ← length(b_points)

    for i in 1, ..., N do
        E_best ← +∞
        for candidate in b_points do
            thresholds ← insertion_sort(sorted_b_points, candidate)
            LI ← get_left_interval(x_sorted, y_sorted, thresholds, candidate)
            RI ← get_right_interval(x_sorted, y_sorted, thresholds, candidate)
            if LI not in interval_cache then
                interval_cache.store(LI, compute_entropy(y_sorted[LI]))
            if RI not in interval_cache then
                interval_cache.store(RI, compute_entropy(y_sorted[RI]))
            E_L ← interval_cache.get(LI)
            E_R ← interval_cache.get(RI)
            E_i ← E_{i-1} − parent_cache.get(candidate) + E_L + E_R
            if E_i < E_best then
                E_best, T_best ← E_i, candidate
                LI_best, RI_best ← LI, RI
                E_{L,best}, E_{R,best} ← E_L, E_R

        append (T_best, E_{i-1} − E_best) to sorted_b_points
        b_points.remove(T_best)
        E_{i-1} ← E_best
        for x in b_points[LI_best] do parent_cache.store(x, E_{L,best})
        for x in b_points[RI_best] do parent_cache.store(x, E_{R,best})
    return sorted_b_points

Procedure get_best_thresholds(b_points_per_feature, N)
    thresholds ← empty list [ ]
    while length(thresholds) < N do
        idx, value ← −1, −∞
        empty ← 0
        for (i, list) in enumerate(b_points_per_feature) do
            if length(list) > 0 then
                if value < list[0][1] then
                    idx ← i
                    value ← list[0][1]
            else
                empty ← empty + 1
        if empty = length(b_points_per_feature) then
            break
        the tuple (idx, value) is appended to thresholds
        the head of b_points_per_feature[idx] is popped
    return thresholds
```

2.1 Considerations about the implementation

When two consecutive observations have different class labels, there are infinitely many possible thresholds that can be chosen to separate them. The function that computes the boundary points returns the average between the two values (line 14). Even if two consecutive observations share the same class label, they can still produce a boundary point. This occurs when at least one other sample, identical to one of the two observations, belongs to a different class (lines 19-22).

Regarding the procedure `sort_boundary_points`, it contains an abuse of notation. Variables LI and RI (lines 35-36) are treated as arrays of indices, but this is problematic as the lists y_{sorted} and b_points (lines 38, 40, 53, 53) are not guaranteed to have the same number of elements. Consequently, using the same indices for both lists is not meaningful. Let S represent the range of values that is being split. LI and RI should be understood as sub-intervals of S, with LI representing the interval preceding the candidate split point and RI representing the interval following it. Therefore, the notation $list[I]$ (where I refers to the same type of interval as RI or LI) should be interpreted as "the subset of $list$ corresponding to the elements in I". Furthermore, it can be shown that theorem 1 can be extended to impurity metrics other than entropy. In this work, the Gini index is considered as an alternative. The function `compute_entropy` (lines 27, 38, 40) can be easily modified accordingly.

Eventually, `get_best_thresholds` produces a global ranking of binary features by merging together the lists returned by `sort_boundary_points`. It also retrieves the top N thresholds in such a ranking. The procedure compares the head of the threshold lists for each feature by entropy gain, and retrieves the best one while removing it (lines 58-67). This can be executed until N binary features are retrieved, or until no more thresholds are available (lines 68, 69). In the resulting ranking, the relative orders of the original features are maintained (Figure 2.2).

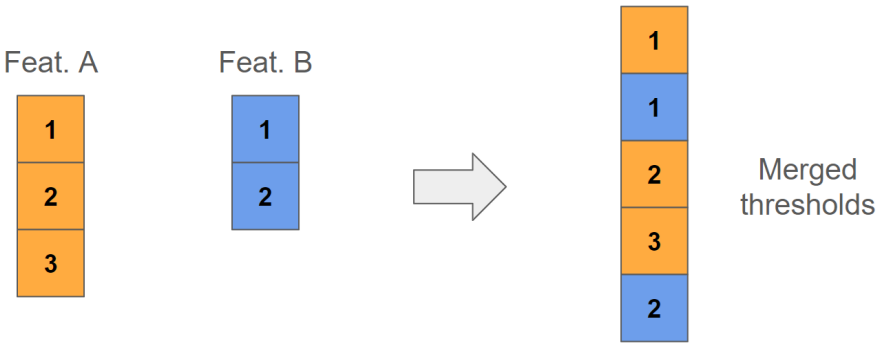


Figure 2.2: The thresholds of all the features are sorted together by entropy gain, but maintaining the original relative orders.

Let N_1, N_2 be two positive integers, and let X_1, X_2 be the sets of thresholds obtained by calling `get_best_thresholds` with parameters N_1 and N_2, respectively. By construction, if $N_1 < N_2$, then $X_1 \subseteq X_2$ making the new set of binary features a superset of the previous one. As explained in section 1.6, this implies that an optimal decision tree trained on X_2 performs as well as, or better than, one trained on X_1. This property could be particularly useful for branch-and-bound algorithms like DL8.5: the error associated with X_1 serves as an upper bound for the error on X_2, and it can be provided to the algorithm at the start of execution to enhance its pruning capabilities. This possibility is further analyzed in Chapter 3.

MID makes it straightforward to represent the observations in a dataset using a small number of binary features. Moreover, additional features can easily be added if the user has more availability in terms of time and resources. This constitutes an advantage with respect to MDLP, whose output has fixed dimensionality. Once the training process is carried out, the actual discretization can be performed by grouping the elements of the *thresholds* array by feature through the *idx* value of each record (line 70), and by applying them on the original attributes of the dataset. Finally, it is important to notice that producing more than one set of binary features starting from the same continuous dataset only requires to train MID a single time.

Chapter 3: Experiments and Results

To evaluate the performance of MID, a series of experiments aimed at answering the following questions have been conducted:

Q1. How does the DL8.5 algorithm perform on continuous data when it is paired with the minimum impurity discretizer? How good is this approach compared with other decision tree learners?

Q2. How does MID behave compared with other discretization techniques?

Q3. As stated at the end of Chapter 2, it is possible to use MID to instantiate the upper bound parameter of DL8.5. Can this effectively speed up its execution?

All experiments were conducted using 14 datasets from the UCI Machine Learning Repository [30] on a cluster with 5120 AMD Epyc Genoa cores. Each node had 766GB of available RAM, running on Rocky Linux version 8.6.

3.1 Experiments about classification performance

The experiments addressing **Q1** and **Q2** compare five classification pipelines: DL8.5 preceded by MID, DL8.5 preceded by the MDLP discretizer[1], CART paired with MID, CART without any discretization strategy, and a model that applies an 8 bins equal-frequency discretization on the data before feeding it to DL8.5. Equal-frequency discretization and MDLP were chosen for comparison with MID

[1]The implementation available at `https://github.com/navicto/Discretization-MDLPC` has been used.

because the former one is commonly used to binarize datasets in the context of itemset mining[2], and because comparative studies [21, 22] have shown that the latter one performs as well as or better than other discretization techniques. CART was chosen among other greedy DT learners because, like DL8.5, it only produces binary decision trees. The classifiers paired with MID were tested multiple times using different numbers of binary input features. All experiments were conducted using 10-folds cross-validation with a maximum depth constraint between 3 and 6. To mitigate overfitting, each leaf of the tree was required to have at least five examples. A time limit of 5 minutes was set for the training of DL8.5. Both entropy and Gini index were used as impurity metrics for MID. In all the experiments, the maximum number of binary features produced by MID was limited to an arbitrary number of 45. This limitation arises from several factors:

- The maximum number of binary features in which a dataset can be discretized depends on the specific observations stored into it, and it can vary considerably depending on the dataset under analysis. All 14 UCI datasets can be discretized into a number of features within this range.

- The number of trees considered by an optimal decision tree algorithm increases exponentially with the number of binary features fed to it. To get a lower bound of such quantity, we can count the number of possible trees having a given depth k and whose levels are completely filled. If N binary features are considered, such number is equal to

$$N_{trees} = \prod_{i=0}^{k-1} (N-i)^{2^i},$$

because each level i contains 2^i nodes, and all the $N - i$ features that did not appear in a higher level can be used in them. For $k = 6$ and $N = 45$, this value is greater than $3 \cdot 10^{101}$. Despite the branch-and-bound and

[2] `https://dtai.cs.kuleuven.be/CP4IM/datasets/`

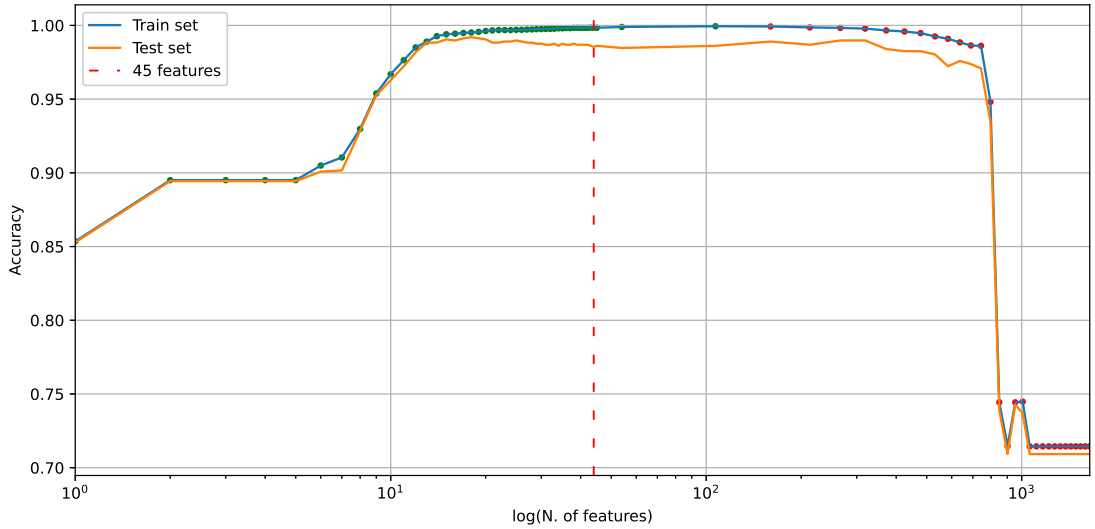


Figure 3.1: Train and test accuracies achieved by MID and DL8.5 on the banknote dataset (maximum depth: 5; impurity metric: Gini index). A classifier was trained for every number of features between 1 and 45, and an additional 32 classifiers were trained using evenly spaced values between 45 and the maximum possible number of features. The green dots correspond to the trees that finished the training process, while the red dots indicate those that timed out. The accuracy of DL8.5 increases only slightly when more than 45 features are provided to it, and drops because of the time out when the input features are more than 100.

caching strategies implemented in DL8.5, if the search space is too large, the algorithm is likely to reach the time limit before finding the optimal solution. The range $[1, 45]$ ensures that most experiments complete the training process within 5 minutes.

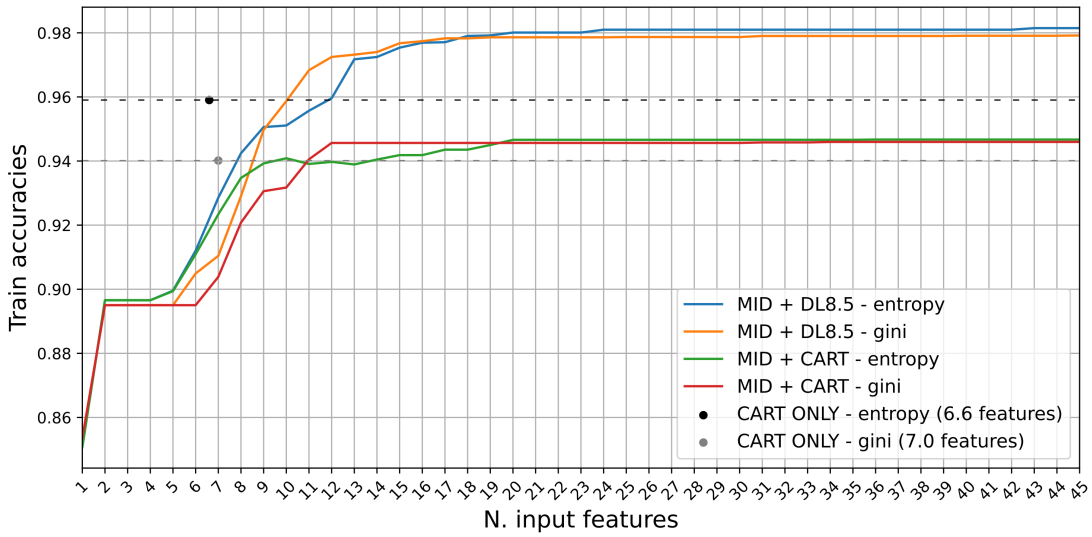
- The trees found using this range of features already achieve high performance both in terms of train and test accuracy, which are comparable with the ones obtained using numbers of features greater than 45 (Figure 3.1).

For now, let us focus on the comparison between DL8.5, preceded by MID, and CART. First, MID is used to extract binary features from a continuous dataset. Then, both DL8.5 and CART are trained on the obtained binary features. CART is also trained on the original continuous dataset. Eventually, the results of the three classifiers are all compared together. Figure 3.2 presents the accuracies and the runtimes achieved on the banknote dataset with a maximum depth of 3. The x-axis represents the number of binary features provided to the classifiers. The

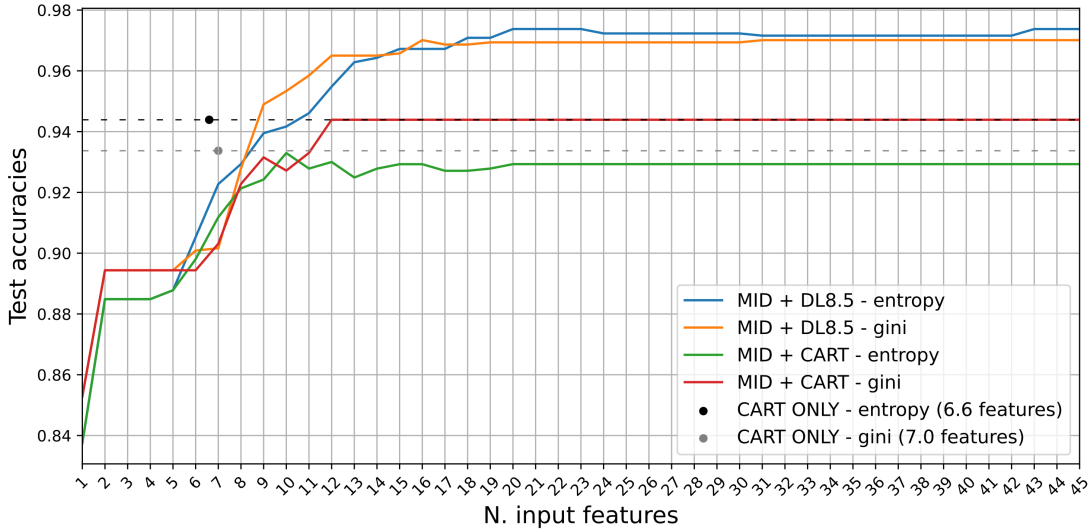
accuracies achieved by CART on continuous data are added as dots using the number of unique thresholds tested in the resulting trees as x-coordinates.

The results of the two DT learners trained on binary data show that DL8.5 achieves better accuracy scores than CART, both on the training set and on the test set. For what concernes the runtimes, instead, even though the two algorithms are comparable when the number of input features is limited, CART demonstrates to be much faster when more features are provided to it. These considerations are coherent with the nature of the two algorithms, as DL8.5 is optimal and CART is greedy. Comparing the performance of DL8.5 with the one achieved by CART on the original continuous data, instead, gives us more information about the quality of the binary features produced by MID. In fact, DL8.5 is not guaranteed to find an optimal tree when the data on top of which it is applied are the result of a discretization process: if the thresholds used in the discretization are poorly chosen, the resulting tree might underperform with respect to a classifier trained on the original continuous dataset. If DL8.5, in conjunction with MID, would achieve a better training accuracy than CART used by itself, it would give us evidence about the efficacy of the features produced by MID. This is exactly what the experiments show: when comparing the number of features actually used in the final trees, as reported in Figure 3.3 for the banknote dataset, it appears that MID and DL8.5 use less features (at most 6) than CART on the continuous data. Thus, they manage to achieve higher accuracies while producing smaller trees.

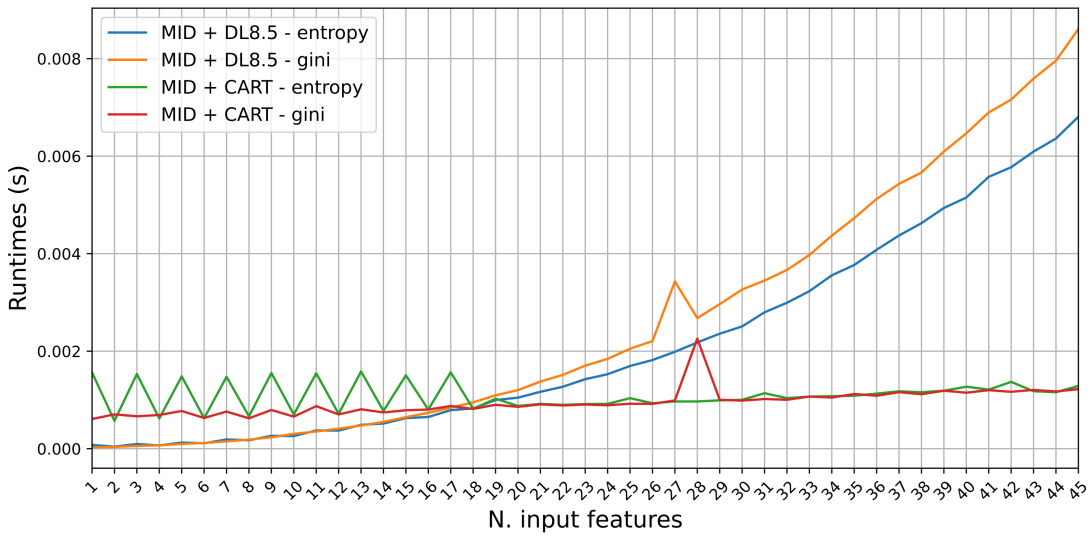
An interesting trend can be observed in the test accuracy of the two algorithms (Figure 3.2b): initially, the accuracies of both CART and DL8.5 increase sharply as the number of binary features produced by MID grows. Both then reach a plateau as the number of features continues to increase. Notably, unlike training accuracy, test accuracy is not guaranteed to be monotonically increasing by design. This suggests that the anytime approach introduced in this work is effective, as providing more training time and features indeed results in better decision trees.



(a) Train accuracies



(b) Test accuracies



(c) Runtimes

Figure 3.2: Results on banknote dataset. Maximum depth of 3.

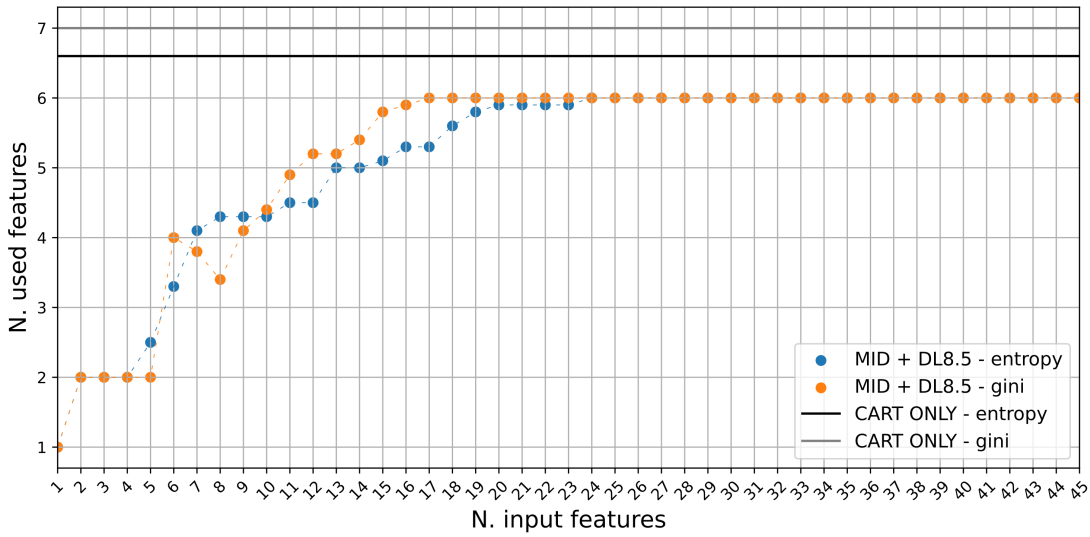


Figure 3.3: Number of unique binary features used in the decision trees found by DL8.5 as a function of the number of features fed to it. Banknote dataset, maximum depth of 3.

The results of the experiments performed on the remaining datasets are coherent with those described above. In most cases, the pipeline composed of MID and DL8.5 outperformed CART trained on continuous data in terms of training set accuracy, using both entropy and Gini index to sort the features. This indicates that the binary features produced by MID provide a good representation of the original datasets. In addition, a similar trend for the test accuracies is observed in most experiments. Notably, after reaching the plateau, the performance on the test set can degrade as more features are provided to the learners. Since the training accuracy cannot decrease, in this case overfitting takes place. This happens particularly in deeper trees (Figure 3.4). This behavior suggests that one could determine how many binary features to use by iteratively adding them until there is no significant change in the test accuracy (i.e., when it stops improving).

Table 3.2 contains the test set accuracies achieved by the five classification pipelines introduced at the beginning of this section, for all the considered datasets and values of maximum depth. For the classifiers paired with the minimum impurity discretizer, the performance obtained after being trained on 45 binary features has been considered. DL8.5 attained the highest accuracy in 20 of the 56 exper-

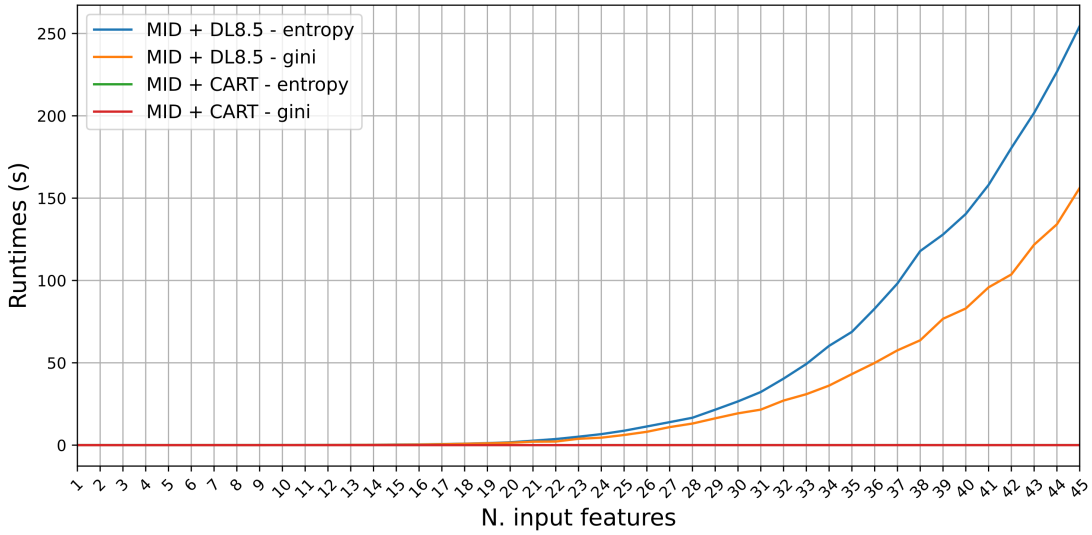
iments when it was paired with MID, using either the entropy or the Gini index to evaluate the goodness of the splits. This is the pipeline that, tied with CART trained on continuous features, achieved the best result in the largest number of experiments. In Table 3.3 the comparison is repeated considering the best possible accuracies obtained by DL8.5 and CART when preceded by MID, across all the numbers of features between 1 and 45. This comparison shows a raise in the number of instances where DL8.5 achieves the best test accuracy to 38 cases.

Dataset	**N. features**	**Accuracy**		**N. features**	**Accuracy**	
		Eq. Freq.	**MID**		**MDLP**	**MID**
Banknote	*28.0*	*96.66*	***98.1***	*12.5*	***95.66***	*95.64*
Breast cancer	*210.0*	*97.46*	***98.05***	*58.3*	***97.95***	***97.95***
Forest covtype	*70.0*	*70.41*	***72.37***	*307.1*	*72.16*	***72.18***
Ionosphere	*205.8*	*93.45*	***94.37***	*99.9*	***94.24***	***94.24***
Iris	*27.7*	*96.67*	***97.78***	*6.78*	***96.38***	***96.38***
Letter recognition	*87.5*	*28.97*	***29.04***	*135.9*	***29.04***	***29.04***
Pendigits	*94.0*	*65.41*	***66.46***	*146.6*	*66.47*	***66.48***
Penguins	*28.0*	*89.52*	***91.59***	*10.6*	***90.46***	*90.39*
Indians	*49.9*	*79.25*	***79.44***	*8.8*	***78.63***	*78.62*
Shuttle	*51.0*	*99.67*	***99.84***	*183.0*	***99.9***	***99.9***
Sonar	*420.0*	*90.97*	***91.56***	*20.2*	***87.13***	*87.07*
Spambase	*73.9*	*88.83*	***90.47***	*97.0*	***90.47***	***90.47***
Wine	*91.0*	*98.69*	***99.38***	*21.8*	***98.94***	*98.88*
Yeast	*39.0*	*55.91*	***58.12***	*11.3*	*57.55*	***57.59***

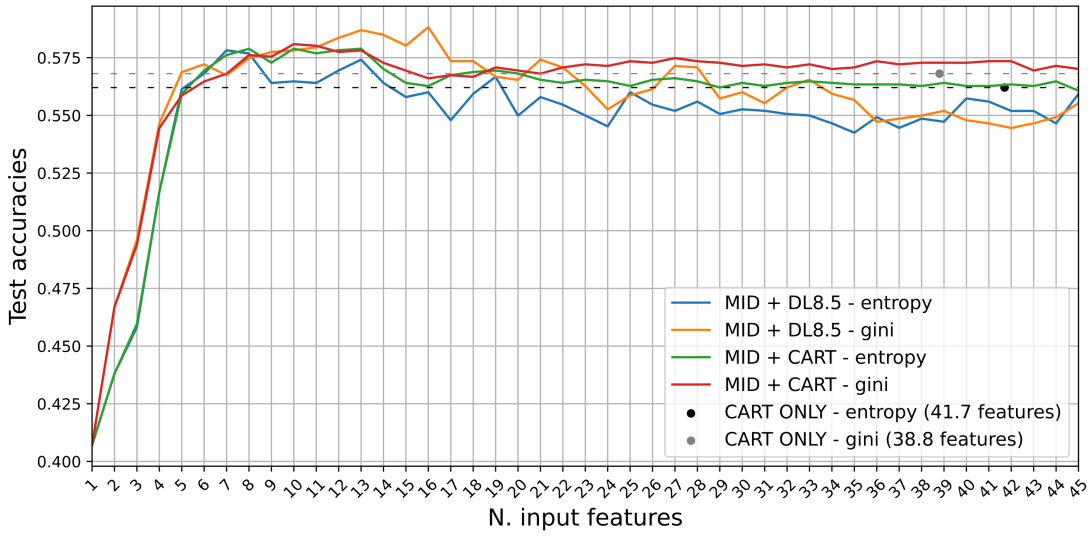
Table 3.1: Train set accuracies achieved by DL8.5 when preceded by three different discretizers: equal-frequency with 8 bins, MDLP and MID. The number of produced binary features is reported

For the sake of completeness, an analysis of the effects of different discretization strategies on the training accuracy of DL8.5 was performed. Table 3.1 contains the results. A maximum depth constraint of 3 was set to ensure that all runs finish without timeout. For every dataset, four sets of binary features were produced: one using 8-bin equal-frequency discretization (X_{EF}), one using the MDLP discretizer (X_{MDLP}), and two using MID to match the number of features in the other sets (X_{MID-EF} for the one matching X_{EF} and $X_{MID-MDLP}$ for the one matching X_{MDLP}). Since the MDLP discretizer is entropy-based, the same impurity metric was used for MID. The classifier trained on X_{MID-EF} consistently

outperforms the one trained on X_{EF}. X_{MDLP} and $X_{MID-MDLP}$ lead to very similar performance, with accuracy scores being identical on 6 out of the 14 datasets and differing by at most 0.07% in the other cases. Moreover, since it is possible to increase the number of features produced by MID, it is also possible that MID could achieve higher scores. For example, Figure 3.2a shows that DL8.5 can achieve a better training accuracy (98%) on the banknote dataset if more than 19 binary features are used.



(a) Runtimes



(b) Test accuracies

Figure 3.4: Test accuracies and runtimes on the yeast dataset for a maximum depth of 6. The accuracies begin to decline after reaching their peak, particularly for DL8.5. Notably, since this decline is not caused by a timeout during the training process, the most likely explanation is overfitting.

DATASET	MAX DEPTH	MID + DL8.5		MID + CART		CART		MDLP + DL8.5	EQ. FREQ. (8 bins) + DL8.5
		Gini	Ent.	Gini	Ent.	Gini	Ent.		
Banknote	3	97.01	**97.38**	94.39	92.93	93.37	94.39	94.46	95.34
	4	**98.32**	**98.32**	95.34	95.48	95.34	95.99	94.75	98.18
	5	98.62	98.83	98.47	97.74	97.01	97.96	94.75	**98.98**
	6	98.69	98.69	98.47	98.18	97.96	97.81	94.75	**99.13**
Breast cancer	3	94.54	**95.43**	94.37	92.97	93.85	93.68	95.07	93.13
	4	94.37	92.43	93.67	92.61	94.03	**94.9**	**94.9**	91.55
	5	92.44	93.5	93.49	93.14	94.56	**95.43**	92.62	91.56
	6	92.44	92.26	94.02	92.96	94.03	**95.26**	91.92	90.33
Forest covtype	3	**68.65**	68.53	67.47	67.06	67.73	67.06	68.62	66.88
	4	**70.32**	70.05	69.19	68.25	70.07	67.92	68.66	67.99
	5	**71.37**	71.07	70.0	70.33	70.23	69.91	68.68	68.47
	6	**72.27**	71.8	71.31	71.27	71.54	71.28	68.64	52.39
Ionosphere	3	86.6	89.17	89.75	**90.89**	89.46	90.32	87.47	88.33
	4	87.47	89.18	86.9	88.6	86.61	87.17	**89.45**	87.77
	5	86.34	89.46	90.03	**92.31**	87.76	88.02	86.07	84.34
	6	86.06	**90.6**	88.33	90.31	87.19	87.75	84.04	83.2
Iris	3	94.67	94.67	**96.67**	**96.67**	**96.67**	**96.67**	94.07	92.67
	4	92.0	94.67	**96.67**	**96.67**	**96.67**	**96.67**	95.83	92.67
	5	94.0	94.67	**96.67**	**96.67**	**96.67**	**96.67**	94.0	92.0
	6	90.67	92.0	**96.67**	**96.67**	**96.67**	**96.67**	94.0	90.67
Letter recognition	3	24.58	24.42	17.72	23.36	17.67	23.12	**25.34**	25.31
	4	37.89	**37.95**	25.56	35.51	25.06	35.28	30.62	30.8
	5	49.3	48.26	36.66	**50.48**	37.24	50.39	30.47	35.21
	6	51.71	51.47	47.29	58.43	48.11	**59.48**	29.57	16.44
Pendigits	3	65.67	**65.98**	56.29	60.2	56.56	60.3	65.89	65.31
	4	82.43	82.3	69.99	77.58	72.4	77.48	**83.91**	83.53
	5	88.71	**89.29**	80.23	83.13	81.9	85.26	74.15	87.21
	6	91.19	**91.73**	85.5	87.66	86.44	90.03	68.39	33.28
Penguins	3	85.88	87.4	85.3	84.71	**89.21**	87.42	86.2	84.06
	4	**90.09**	89.48	84.69	85.0	88.29	88.9	87.42	84.71
	5	88.0	86.78	84.98	87.09	**89.79**	88.89	88.62	84.44
	6	88.0	89.19	86.17	85.57	**89.78**	88.89	87.42	84.41
Indians	3	73.82	74.09	75.26	74.6	**75.39**	74.73	73.3	74.99
	4	**75.77**	74.47	74.21	74.21	73.04	72.52	74.21	74.87
	5	73.83	75.64	76.03	76.16	**76.81**	75.38	74.6	74.35
	6	72.4	73.32	74.87	**76.69**	74.48	73.18	74.6	69.55
Shuttle	3	99.71	99.8	99.61	99.6	99.64	99.64	**99.87**	99.64
	4	99.85	**99.91**	99.76	99.75	99.8	99.8	99.9	99.72
	5	99.9	**99.93**	99.83	99.88	99.89	**99.93**	99.85	99.74
	6	99.91	99.92	99.88	99.89	99.92	**99.96**	99.87	99.74
Sonar	3	69.26	69.86	67.86	69.29	68.9	70.21	67.29	**74.52**
	4	73.57	74.05	72.67	66.86	69.81	**77.98**	76.05	74.12
	5	75.62	71.21	**76.48**	70.21	71.26	74.55	76.02	71.71
	6	71.64	69.74	76.48	71.21	69.81	**76.5**	66.86	72.1
Spambase	3	88.76	89.02	87.66	86.46	88.13	86.29	**90.22**	88.65
	4	90.05	90.02	89.59	89.5	89.48	89.96	**91.33**	89.24
	5	91.02	91.4	91.07	91.41	90.35	90.48	**91.98**	89.13
	6	**92.7**	92.31	91.98	91.44	91.04	91.2	91.11	89.13
Wine	3	**94.93**	94.44	87.65	92.74	89.34	93.3	93.33	93.82
	4	92.12	91.08	87.65	92.74	89.34	**93.3**	93.17	89.9
	5	92.09	**94.41**	87.65	92.74	89.34	93.3	93.24	91.08
	6	92.19	92.16	87.65	92.74	89.34	**93.3**	92.71	89.77
Yeast	3	55.32	**57.08**	54.92	50.74	55.12	50.74	56.27	54.04
	4	56.88	56.4	55.93	56.41	55.53	56.0	57.62	**58.15**
	5	54.85	56.67	56.81	56.68	**57.21**	56.0	56.74	54.72
	6	55.53	55.93	**57.01**	56.07	56.81	56.2	56.4	55.19

Table 3.2: Test accuracies achieved by the 5 considered classification pipelines, on all the datasets and for all the tested values of maximum depth. The four columns associated to MID contain the results attained by the classifiers when trained on 45 binary features.

DATASET	MAX DEPTH	MID + DL8.5		MID + CART		CART		MDLP + DL8.5	EQ. FREQ. (8 bins) + DL8.5
		Gini	Ent.	Gini	Ent.	Gini	Ent.		
Banknote	3	97.01	**97.38**	94.39	93.29	93.37	94.39	94.46	95.34
	4	**99.05**	98.98	95.7	96.57	95.34	95.99	94.75	98.18
	5	99.2	**99.27**	98.91	98.1	97.01	97.96	94.75	98.98
	6	99.2	**99.27**	98.91	98.25	97.96	97.81	94.75	99.13
Breast cancer	3	95.08	**95.43**	94.55	94.38	93.85	93.68	95.07	93.13
	4	**94.91**	94.2	94.38	93.67	94.03	94.9	94.9	91.55
	5	94.9	95.08	94.38	94.2	94.56	**95.43**	92.62	91.56
	6	**95.43**	95.08	94.9	94.2	94.03	95.26	91.92	90.33
Forest covtype	3	**68.69**	68.53	67.49	67.54	67.73	67.06	68.62	66.88
	4	**70.32**	70.05	69.19	68.44	70.07	67.92	68.66	67.99
	5	**71.69**	71.23	70.0	70.33	70.23	69.91	68.68	68.47
	6	**72.33**	71.83	71.31	71.27	71.54	71.28	68.64	52.39
Ionosphere	3	90.03	**91.75**	90.32	90.89	89.46	90.32	87.47	88.33
	4	90.03	**91.75**	90.32	90.89	86.61	87.17	89.45	87.77
	5	90.31	91.75	90.89	**92.31**	87.76	88.02	86.07	84.34
	6	90.32	92.02	90.32	**92.31**	87.19	87.75	84.04	83.2
Iris	3	96.0	96.0	**96.67**	**96.67**	**96.67**	**96.67**	94.07	92.67
	4	94.67	95.33	**96.67**	**96.67**	**96.67**	**96.67**	95.83	92.67
	5	94.67	94.67	**96.67**	**96.67**	**96.67**	**96.67**	94.0	92.0
	6	94.67	94.67	**96.67**	**96.67**	**96.67**	**96.67**	94.0	90.67
Letter recognition	3	24.58	24.44	17.76	23.36	17.67	23.12	**25.34**	25.31
	4	**38.21**	38.2	25.58	35.51	25.06	35.28	30.62	30.8
	5	52.17	**53.26**	36.7	50.55	37.24	50.39	30.47	35.21
	6	58.5	**60.55**	47.29	58.43	48.11	59.48	29.57	16.44
Pendigits	3	65.68	**66.05**	56.3	60.8	56.56	60.3	65.89	65.31
	4	82.43	82.33	70.24	77.58	72.4	77.48	**83.91**	83.53
	5	88.71	**89.29**	80.23	83.52	81.9	85.26	74.15	87.21
	6	92.02	**92.68**	85.5	88.2	86.44	90.03	68.39	33.28
Penguins	3	88.31	**89.5**	87.99	86.51	89.21	87.42	86.2	84.06
	4	**90.4**	**90.4**	88.31	89.21	88.29	88.9	87.42	84.71
	5	90.7	**91.59**	88.31	88.61	89.79	88.89	88.62	84.44
	6	90.7	**91.01**	88.31	88.61	89.78	88.89	87.42	84.41
Indians	3	75.26	**75.52**	75.26	74.6	75.39	74.73	73.3	74.99
	4	**76.17**	75.65	75.51	75.65	73.04	72.52	74.21	74.87
	5	76.68	75.91	76.69	76.42	**76.81**	75.38	74.6	74.35
	6	75.52	75.91	75.91	**76.82**	74.48	73.18	74.6	69.55
Shuttle	3	99.71	99.8	99.61	99.6	99.64	99.64	**99.87**	99.64
	4	99.85	**99.91**	99.77	99.78	99.8	99.8	99.9	99.72
	5	99.9	**99.93**	99.84	99.88	99.89	**99.93**	99.85	99.74
	6	99.91	99.92	99.89	99.89	99.92	**99.96**	99.87	99.74
Sonar	3	**76.02**	72.71	72.67	73.71	68.9	70.21	67.29	74.52
	4	77.95	**78.88**	77.02	75.12	69.81	77.98	76.05	74.12
	5	75.62	76.48	**77.48**	76.1	71.26	74.55	76.02	71.71
	6	77.93	76.1	**78.0**	75.62	69.81	76.5	66.86	72.1
Spambase	3	88.83	89.02	87.89	86.83	88.13	86.29	**90.22**	88.65
	4	90.65	90.28	89.72	89.76	89.48	89.96	**91.33**	89.24
	5	91.37	91.57	91.33	91.57	90.35	90.48	**91.98**	89.13
	6	**92.7**	92.35	92.13	91.55	91.04	91.2	91.11	89.13
Wine	3	95.52	**96.11**	89.31	93.3	89.34	93.3	93.33	93.82
	4	95.55	**95.56**	89.38	92.74	89.34	93.3	93.17	89.9
	5	94.96	**96.08**	89.38	92.74	89.34	93.3	93.24	91.08
	6	95.55	**96.08**	89.38	92.74	89.34	93.3	92.71	89.77
Yeast	3	55.79	**57.08**	55.39	50.74	55.12	50.74	56.27	54.04
	4	**58.76**	58.56	57.21	56.74	55.53	56.0	57.62	58.15
	5	**58.42**	58.02	57.88	58.16	57.21	56.0	56.74	54.72
	6	**58.83**	57.82	58.09	57.89	56.81	56.2	56.4	55.19

Table 3.3: Test accuracies achieved by the 5 considered classification pipelines, on all the datasets and for all the tested values of maximum depth. The four columns associated to MID contain the best results attained by the classifiers among all the tested numbers of binary features.

3.2 Experiments about runtime

Before describing how **Q3** has been addressed, let us first have a better understanding of the problem at hand. As previously introduced in section 3.1, the size of the dataset fed to DL8.5 has an impact on the time needed for its training process, and consequently on its accuracy if a time limit is specified. As reported in [31], since the algorithm uses a depth-first search to explore the search space of decision trees, stopping the execution before its completion can lead to an unbalanced classifier: not enough time may have been dedicated to refining the right-hand branches, potentially resulting in a tree "leaning to the left" (Figure 3.5), characterized by a substantial classification error.

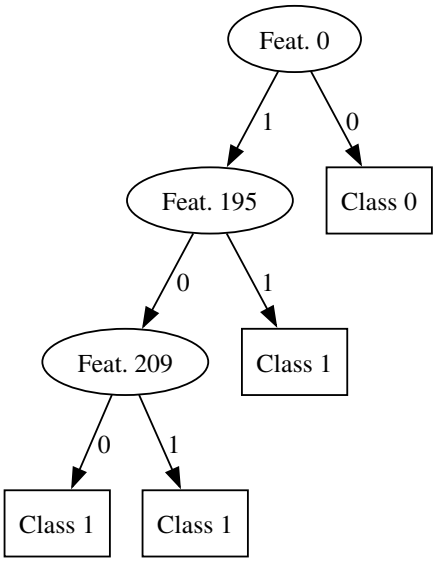


Figure 3.5: Result of DL8.5 tree over the banknote dataset using a maximum depth of 5 and 1300 binary input features, after the 5 minutes timeout.

The objective of this set of experiments was to investigate the potential use of the Minimum Impurity Discretizer to accelerate the execution of DL8.5 and mitigate the problem described above. In section 1.3 we have seen that DL8.5 uses a branch and bound approach to exclude some of the solutions at runtime, and that it keeps track of an upper bound on the quality of the trees to do so. If no reference of the performance that should be expected from the classifier is

available before running DL8.5, the upper bound is initialized to $+\infty$ and updated during the execution. However, if such an estimate is available, more branches can potentially be pruned. The idea is to use MID to generate two sets of data X_l and X_h starting from the same continuous dataset, containing N_l and N_h attributes, respectively. Assuming $N_h > N_l$, X_h is a superset of X_l, and we already saw in Section 2.1 that the error associated to dataset X_l can be used as upper bound for the execution of DL8.5 over X_h. Two types of experiment have been conducted to test this strategy. These are described and analyzed below. Figures 3.6 and 3.7 show the results obtained on the banknote and iris datasets, respectively.

Type a. In Figures 3.6a and 3.7a, each column corresponds to a classifier trained on a dataset comprising 45 binary features, using as upper bound the training error of another classifier previously trained on a smaller set of data. For instance, to generate the column at index 10 on the x-axis, DL8.5 was initially run on a discretized version of the banknote dataset containing 10 features. Subsequently, the training error from this run served as an upper bound to train another classifier using 45 features. The green columns depict the runtimes of DL8.5 when executed over the 45-features dataset using the upper bound, while the blue ones show the runtimes required to train the classifiers on the smaller datasets. The red line represents the baseline, namely the runtime needed to run DL8.5 over 45 features without providing any upper bound.

Type b. Regarding Figures 3.6b and 3.7b, instead, two columns are depicted for each number of features N_i reported on the x-axis. The blue column indicates the time required to train a classifier on a dataset containing N_i features, with no upper bound on the training error. The orange column, instead, represents the runtime of DL8.5 on the same dataset when an initial upper bound is set. Let $t_{u,i}$ and $t_{b,i}$ denote the times needed to train the $i-$th unbounded and bounded

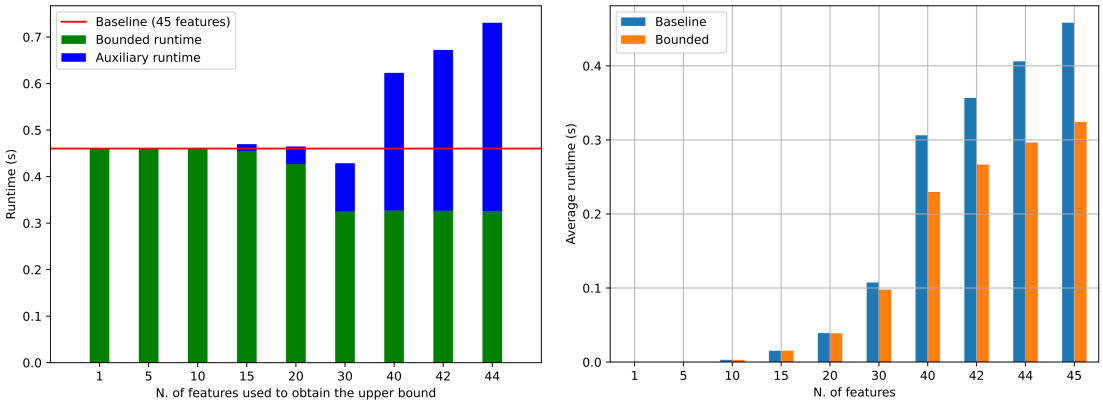
classifiers, respectively, and let e_i be the associated training error. The upper bound used for the tree corresponding to the runtime $t_{b,i}$ is e_{i-1}.

If an initial upper bound is provided to DL8.5, and no tree in the search space achieves a training error strictly lower than it, then no classifier is returned as a result. For both types of experiments, in this scenario the training process is repeated with an upper bound increased by 0.1%. The number of features tested in these experiments remains limited between 1 and 45 for the same reasons outlined in section 3.1. The duration of the training of each classifier is particularly crucial in this scenario, as problems arise both if it is excessively long or excessively short:

- In the former case, the experiments become meaningless if DL8.5 reaches the timeout, as no differences between runtimes in different configurations can be observed.

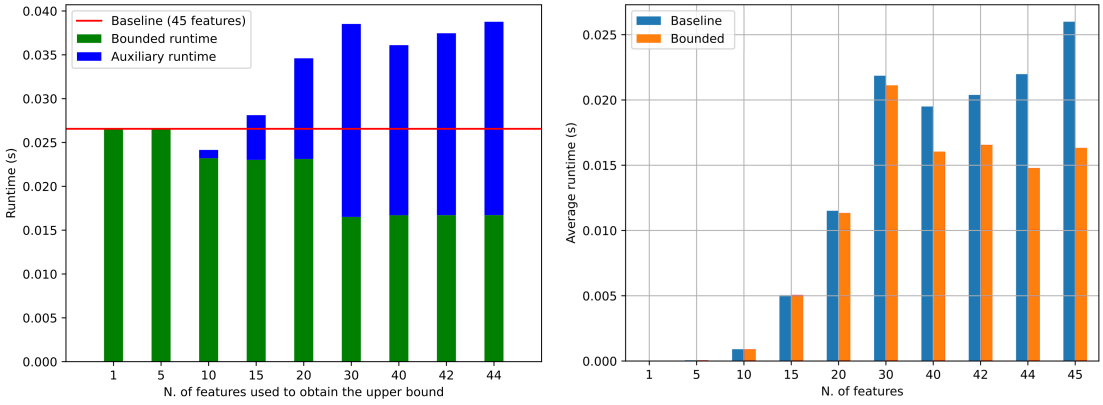
- In the latter case, obtaining a reliable estimate of the runtime of DL8.5 becomes challenging, as measurements of shorter time intervals are more susceptible to fluctuations. To mitigate the effect of such fluctuations, each runtime has been computed as the average of 1000 measurements performed under the same conditions.

The maximum depth parameter of DL8.5 has been individually set for every dataset to achieve the best trade-off between the effects mentioned above. Since classification performance is not of interest in this case, every classifier has been trained on the entire dataset each time. The Minimum Impurity Discretizer has been configured to use entropy as the impurity measure in all experiments.



(a) (b)

Figure 3.6: Runtimes results for the banknote dataset. Maximum depth: 5. Impurity metric: entropy.



(a) (b)

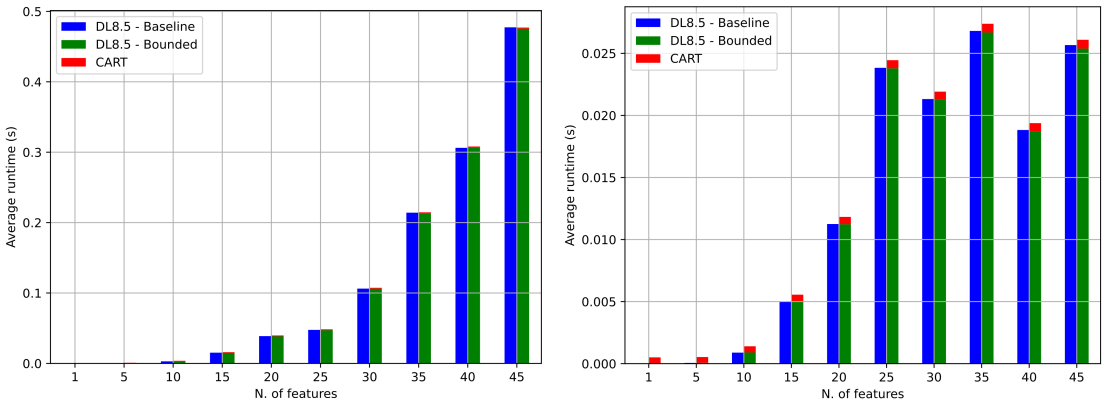
Figure 3.7: Runtimes results for the iris dataset. Maximum depth: 5. Impurity metric: entropy.

Figures 3.6 and 3.7 show that while the methodologies described above can indeed reduce the runtime of DL8.5, applying them is actually not helpful: in most cases, the time saved by providing an upper bound is equal to or less than the time required to compute it. There are some exceptions to this trend, but they occur only in datasets that already have short training times, where further reductions are not very beneficial.

Since the main issue with this strategy appears to be the time required to estimate the upper bound, a potential alternative is to use a faster algorithm to do so. This is the approach that was applied to obtain the results shown in Figure

3.8. For each N_i reported on the x-axis, an instance of CART was trained on a binary dataset containing N_i features. Then, an instance of DL8.5 was trained on the same dataset, using the error achieved by CART as the upper bound. Since both algorithms use the same dataset and DL8.5 is optimal, its training error is guaranteed to be equal to or lower than that of CART. The green bar represents the runtime of CART when the upper bound is specified, while the red one corresponds to the runtime of CART. The blue line serves as baseline, indicating the runtime of DL8.5 when no upper bound is specified.

Unfortunately, this strategy also does not work as expected: although the time required to estimate the upper bound is significantly shorter than before, the training error of CART is too large to effectively prune the search space of the solutions. As a result, the runtime of DL8.5 remains almost unchanged.



(a) Banknote dataset (b) Iris dataset

Figure 3.8: Runtimes results obtained using CART to compute the upper bound for DL8.5. Maximum depth: 5. Impurity metric: entropy.

The results of the experiments on the remaining datasets are consistent with those described above.

Chapter 4: Concluding Remarks

4.1 Summary

Since finding an optimal decision tree under constraints is NP-hard, greedy algorithms like CART have long been the preferred methods to accomplish this task. However, optimal algorithms do exist, and they often require continuous features to be discretized before the learning process, as they are typically designed for binary data. The choice of discretization strategy significantly affects the performance of ODT algorithms: the binarized data can become excessively large, leading to long search times, and the resulting trees are not guaranteed to be optimal. In this work, MID is introduced. This is a new supervised, heuristic-based discretization technique developed to improve the performance on continuous datasets of ODT algorithms such as DL8.5. MID extracts a user-specified number of binary features from a continuous dataset by iteratively splitting in two the range of values of continuous features. It uses an impurity metric to select at every iteration both which is the best feature to split and where to apply the cut. If MID is used to produce two sets of binary features starting from the same continuous dataset, the larger one is a superset of the smaller one. This allows an ODT algorithm to be run repeatedly on input data with a growing dimensionality, producing trees with a growing training accuracy. The advantage of this strategy is that if we run it without time limit, an ODT can be found, but the search process can also be interrupted at any time for a smaller number of features. Experiments on 14 continuous datasets show how in most cases DL8.5 is able to

achieve better performance when trained on a small number of features produced by MID rather than on the output of other discretization techniques, and that it is able to outperform the greedy algorithm CART when this is trained on the original continuous datasets. Experiments have also been conducted to explore the possibility of using MID properties to initialize the upper bound parameter of DL8.5, and improve the efficacy of its branch and bound approach. These experiments, however, gave negative results: in most cases, the time saved by specifying the upper bound was smaller than the time needed to estimate it.

4.2 Future directions

As a future work, it could be valuable to develop an MDL criterion similar to the one employed by MDLP. This enhancement would simplify the fine-tuning of MID by providing users with an initial set of features to begin with. Additionally, the effect of other impurity metrics on the performance of MID and DL8.5 can also be explored. While the experiments aimed at improving the speed of DL8.5 by specifying an upper bound were not successful, there remain other promising avenues to explore. For instance, alternative algorithms, both greedy and optimal, beyond CART and DL8.5, could be investigated to estimate the upper bound. Finally, it would be interesting to use the anytime approach that combines MID and ODT algorithms to solve other tasks besides classification (e.g. regression).

REFERENCES

[1] L. Breiman, *Classification and regression trees*. Routledge, 2017.

[2] J. R. Quinlan, *C4. 5: programs for machine learning*. Elsevier, 2014.

[3] H. Liu, F. Hussain, C. L. Tan, and M. Dash, "Discretization: An enabling technique," *Data mining and knowledge discovery*, vol. 6, pp. 393–423, 2002.

[4] J. Dougherty, R. Kohavi, and M. Sahami, "Supervised and unsupervised discretization of continuous features," in *Machine learning proceedings 1995*. Elsevier, 1995, pp. 194–202.

[5] G. Aglin, S. Nijssen, and P. Schaus, "Learning optimal decision trees using caching branch-and-bound search," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 34, no. 04, 2020, pp. 3146–3153.

[6] S. Aghaei, A. Gómez, and P. Vayanos, "Strong optimal classification trees," *Operations Research*, 2024.

[7] S. Nijssen and E. Fromont, "Mining optimal decision trees from itemset lattices," in *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2007, pp. 530–539.

[8] E. Demirović, A. Lukina, E. Hebrard, J. Chan, J. Bailey, C. Leckie, K. Ramamohanarao, and P. J. Stuckey, "Murtree: Optimal decision trees via dynamic programming and search," *Journal of Machine Learning Research*, vol. 23, no. 26, pp. 1–47, 2022.

[9] J. Lin, C. Zhong, D. Hu, C. Rudin, and M. Seltzer, "Generalized and scalable optimal sparse decision trees," in *International Conference on Machine Learning*. PMLR, 2020, pp. 6150–6160.

[10] H. Verhaeghe, S. Nijssen, G. Pesant, C.-G. Quimper, and P. Schaus, "Learning optimal decision trees using constraint programming," *Constraints*, vol. 25, pp. 226–250, 2020.

[11] S. Verwer and Y. Zhang, "Learning optimal classification trees using a binary linear program formulation," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 33, no. 01, 2019, pp. 1625–1632.

[12] D. Bertsimas and J. Dunn, "Optimal classification trees," *Machine Learning*, vol. 106, pp. 1039–1082, 2017.

[13] L. Breiman, "Random forests," *Machine learning*, vol. 45, pp. 5–32, 2001.

[14] P. Vitányi, *Computational Learning Theory: Second European Conference, EuroCOLT'95, Barcelona, Spain, March 13-15, 1995. Proceedings*. Springer Science & Business Media, 1995, vol. 2.

[15] M. Fratello, R. Tagliaferri *et al.*, "Decision trees and random forests," *Encyclopedia of bioinformatics and computational biology: ABC of bioinformatics*, vol. 1, no. S 3, 2018.

[16] J. R. Quinlan, "Induction of decision trees," *Machine learning*, vol. 1, pp. 81–106, 1986.

[17] J. J. Boutilier, C. Michini, and Z. Zhou, "Shattering inequalities for learning optimal decision trees," in *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*. Springer, 2022, pp. 74–90.

[18] N. Narodytska, A. Ignatiev, F. Pereira, and J. Marques-Silva, "Learning optimal decision trees with sat," in *International Joint Conference on Artificial Intelligence 2018*. Association for the Advancement of Artificial Intelligence (AAAI), 2018, pp. 1362–1368.

[19] S. Kotsiantis and D. Kanellopoulos, "Discretization techniques: A recent survey," *GESTS International Transactions on Computer Science and Engineering*, vol. 32, no. 1, pp. 47–58, 2006.

[20] R. Dash, R. L. Paramguru, and R. Dash, "Comparative analysis of supervised and unsupervised discretization techniques," *International Journal of Advances in Science and Technology*, vol. 2, no. 3, pp. 29–37, 2011.

[21] S. Garcia, J. Luengo, J. A. Sáez, V. Lopez, and F. Herrera, "A survey of discretization techniques: Taxonomy and empirical analysis in supervised learning," *IEEE transactions on Knowledge and Data Engineering*, vol. 25, no. 4, pp. 734–750, 2012.

[22] Y. Kaya and R. Tekin, "Comparison of discretization methods for classifier decision trees and decision rules on medical data sets," *Avrupa Bilim ve Teknoloji Dergisi*, no. 35, pp. 275–281, 2022.

[23] J. Catlett, "On changing continuous attributes into ordered discrete attributes," in *Machine Learning—EWSL-91: European Working Session on Learning Porto, Portugal, March 6–8, 1991 Proceedings 5*. Springer, 1991, pp. 164–178.

[24] M. Hacıbeyoğlu and M. H. Ibrahım, "Comparison of the effect of unsupervised and supervised discretization methods on classification process," *International Journal of Intelligent Systems and Applications in Engineering*, vol. 4, no. Special Issue-1, pp. 105–108, 2016.

[25] T. H. A. Tran, M. L. Wiesner, and M. van Keulen, "Influence of discretization granularity on learning classification models," in *BNAIC/BeNeLearn 2022 Joint International Scientific Conferences on AI and Machine Learning*, 2022.

[26] J. Y. Ching, A. K. C. Wong, and K. C. C. Chan, "Class-dependent discretization for inductive learning from continuous and mixed-mode data," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 17, no. 7, pp. 641–651, 1995.

[27] J. L. Flores, I. Inza, and P. Larrañaga, "Wrapper discretization by means of estimation of distribution algorithms," *Intelligent Data Analysis*, vol. 11, no. 5, pp. 525–545, 2007.

[28] R. Kerber, "Chimerge: Discretization of numeric attributes," in *Proceedings of the tenth national conference on Artificial intelligence*, 1992, pp. 123–128.

[29] U. M. Fayyad and K. B. Irani, "Multi-interval discretization of continuous-valued attributes for classification learning," in *Ijcai*, vol. 93, no. 2. Citeseer, 1993, pp. 1022–1029.

[30] K. N. Markelle Kelly, Rachel Longjohn, "The UCI Machine Learning Repository," https://archive.ics.uci.edu, [Last accessed 2024/05/28].

[31] H. Kiossou, P. Schaus, S. Nijssen, and V. R. Houndji, "Time constrained dl8. 5 using limited discrepancy search," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 2022, pp. 443–459.