arXiv version 1 (2020) 1-48 Submitted 4/00; Published 10/00

MurTree: Optimal Classification Trees via Dynamic Programming and Search

Emir Demirović E.DEMIROVIC@TUDELFT.NL
Delft University of Technology
Delft, The Netherlands

Anna Lukina ANNA.LUKINA@IST.AC.AT
Institute of Science and Technology Austria
Klosterneuburg, Austria

Emmanuel Hebrard HEBRARD@LAAS.FR
LAAS CNRS
Toulouse, France

Jeffrey Chan JEFFREY.CHAN@RMIT.EDU.AU
RMIT University
Melbourne, Australia

James Bailey BAILEYJ@UNIMELB.EDU.AU
Christopher Leckie CALECKIE@UNIMELB.EDU.AU
Kotagiri Ramamohanarao KOTAGIRI@UNIMELB.EDU.AU
University of Melbourne
Melbourne, Australia

Peter J. Stuckey PETER.STUCKEY@MONASH.EDU
Monash University
Melbourne, Australia

Editor: Editors

Abstract

Decision tree learning is a widely used approach in machine learning, favoured in applications that require concise and interpretable models. Heuristic methods are traditionally used to quickly produce models with reasonably high accuracy. A commonly criticised point, however, is that the resulting trees may not necessarily be the best representation of the data in terms of accuracy, size, and other considerations such as fairness. In recent years, this motivated the development of optimal classification tree algorithms that globally optimise the decision tree in contrast to heuristic methods that perform a sequence of locally optimal decisions. We follow this line of work and provide a novel algorithm for learning optimal classification trees based on dynamic programming and search. Our algorithm supports constraints on the depth of the tree and number of nodes and we argue it can be extended with other requirements. The success of our approach is attributed to a series of specialised techniques that exploit properties unique to classification trees. Whereas algorithms for optimal classification trees have traditionally been plagued by high runtimes and limited scalability, we show in a detailed experimental study that our approach uses only a fraction of the time required by the state-of-the-art and can handle datasets with tens of thousands of instances, providing several orders of magnitude improvements and notably contributing towards the practical realisation of optimal decision trees.

Keywords: Decision trees, optimality, constraints, search, dynamic programming

1. Introduction

Decision trees are traditionally built using heuristic methods, such as CART (Breiman et al. (1984)), and can produce high-quality trees in low computational time. A commonly criticised point, however, is that heuristically constructed decision trees may not necessarily be the best representation of the data in terms of accuracy, size, or other considerations such as fairness.

An alternative is to construct *optimal decision trees*, i.e., the best possible decision tree according to a given metric. The idea of computing optimal decision trees dates back to approximately the 1970s when constructing optimal decision trees was proven to be **NP**-hard by Laurent and Rivest (1976). As emphasised by Bertsimas and Dunn (2017), while optimal decision trees have always been desirable, the authors of the CART algorithm (Breiman et al. (1984)) found that such trees were computationally infeasible given the resources of the time, and hence heuristic algorithms were the only option.

Optimal decision trees are enticing for several reasons. It has been observed that a more accurate representation of the data offers better generalisation on unseen data (Bertsimas and Dunn (2017); Verwer and Zhang (2017, 2019)). This has been reiterated in our experiments as well. Optimal decision trees allow incorporating additional constraints that may be difficult to support in a heuristic algorithm. This is particularly important in socially-sensitive contexts, where special measures need to be taken to ensure *fairness* in machine learning. Otherwise, obtained models may implicitly or explicitly perpetuate discrimination and biases, reducing social welfare (Aghaei et al. (2019)). In some applications, the goal is to optimise the size of the decision tree representing a given controller to save memory for embedded devices (Ashok et al. (2020)). Decision trees, in particular those of small size, are desirable for formal methods when verifying properties of trained controllers, as opposed to more complex machine learning models (Bastani et al. (2018)). In recent years, there has been growing interest in *explainable artificial intelligence*. The basic premise is that machine learning models, apart from high accuracy, must also be able to explain their decisions to a (non-expert) human. This is necessary to increase human trust and reliability of machine learning in complex scenarios that are conventionally handled by humans. Optimal decision trees of small size naturally fit within the scope of explainable AI, as their reduced size is more convenient for human interpretation.

Learning problems are defined as mathematical programs: an objective function is posed possibly together with a set of constraints. An advantage of optimal decision tree algorithms over heuristic approaches is that they adhere precisely to the given specification. This allows a clear analysis and assessment of the suitability of the particular mathematical formulation for a given application. In contrast, in heuristic methods there is a discrepancy between the target learning problem and the goals of the heuristic algorithm. In more detail, heuristic methods for decision trees do not necessarily directly optimise according to the learning problem, but rather locally optimise a sequence of subproblems with respect to a surrogate metric. While this has shown to produce reasonably accurate models quickly, it may be difficult to make conclusive statements on the learning problem definition, as the heuristic approach may not faithfully follow the desired metrics. For example, a specification might

be deemed suboptimal not due to a flaw in the definition, but rather because of the inability of the heuristic algorithm to optimise according to the specification.

Despite the appeal of optimal algorithms for decision trees, heuristic methods are historically the dominant approach due to computational reasons. As both algorithmic techniques and hardware advanced, optimal decision trees have become within practical reach and attracted growing interest from the research community. In particular, there has been a surge of successful methods in the past few years. These approaches use generic optimisation methods, namely integer programming (Bertsimas and Dunn (2017); Verwer and Zhang (2017, 2019); Aghaei et al. (2019)), constraint programming (Verhaeghe et al. (2019)), and SAT (Narodytska et al. (2018)), and algorithms tailored to the decision tree problem (Nijssen and Fromont (2007); Hu et al. (2019); Aglin et al. (2020)). The methods DL8 (Nijssen and Fromont (2007)) and DL8.5 (Aglin et al. (2020)) are of particular interest as they can be seen as a starting point for our work. The DL8.5 approach has been shown to be highly effective, outperforming the other approaches, and is a demonstration that specialised methods may have an advantage over generic optimisation.

Our Contribution. While previous works use highly related ideas, the presentation and terminology may differ substantially. In this work, we unify and generalise successful concepts from the literature by viewing the problem through the lens of a conventional algorithmic framework, namely dynamic programming and search. We introduce novel algorithmic techniques that reduce computation time by orders of magnitude when compared to the state-of-the-art. This notably contributes towards the practical application of optimal classification trees, which was traditionally plagued by high runtimes. We conduct an experimental study on a wide range of benchmarks from the literature to show the effectiveness of our approach and its components, and reiterate that optimal decision trees lead to better generalisation in terms of out-of-sample accuracy. Our framework supports constraints on the depth of the tree and the number of nodes, and we argue it is flexible and may be extended with other requirements. In more detail, the contributions are as follows:

- *MurTree* (Section 4), a framework for computing optimal classification trees, i.e., decision trees that minimise the number of misclassifications. The framework allows constraints on the depth and the number of nodes of the decision tree. The node constraint is not considered in all works on optimal decision trees, notably it is not supported by the previously fastest algorithm, *DL8.5*. Additional objective functions and constraints may be added that admit a dynamic programming formulation (Section 4.9).

- A clear high-level view of the framework using conventional algorithmic principles, namely dynamic programming and search, that unifies and generalises ideas from the literature (Section 4.1).

- A specialised algorithm for computing the optimal classification tree of depth two, which serves as the backbone of our framework (Section 4.3). It uses a frequency counting method to avoid explicitly referring to the dataset. This substantially reduces the runtime of computing optimal trees which, when combined with an incremental technique that takes into account previous computations, provides orders of magnitude speed-ups.

- A novel similarity-based lower bound on the number of misclassifications for an optimal decision tree. The bound is effective in determining that portions of the search space cannot contain better decision trees than currently found during the search, which allows the algorithm to prune parts of the search space without needing further inspection, providing additional speed-ups. The bound is derived by examining previously computed subtrees and computing the number of misclassifications that must hold in the new search space (Section 4.5).

- We incorporate the constraint on the number of nodes in the tree, extend the caching technique to take into account both the depth and number of nodes constraint (Section 4.6), refine the lower bounding technique on the number of misclassifications from DL8.5 (Aglin et al. (2020)) to produce stronger bounds (4.6.1), and provide an incremental solving option to allow reusing computations when solving a series of increasingly large decision trees (Section 4.6.3), e.g., as encountered in hyper-parameter tuning. Further improvements include a dynamic post-order node exploration strategy (Section 4.7) that leads to consistent improvements over a conventional post-order search.

- We provide a detailed experimental study to analyse the effectiveness of our individual techniques and scalability of our approach, evaluate our approach with respect to the state-of-the-art optimal classification tree algorithms, and compare against heuristic decision tree and random forest algorithms on out-of-sample accuracy (Section 5). The experimental results show that our approach provides highly accurate trees and exhibits speed-ups of (several) orders of magnitude when compared to the state-of-the-art.

The rest of the paper is organised as follows. In the next section, we introduce the notions and definitions used throughout the paper. In Section 3, we review the state-of-the-art for optimal decision trees. Our main contribution is given in Section 4, where we describe our *MurTree* framework. In Section 5, we conduct a series of empirical evaluations of our approach and conclude in Section 6.

2. Preliminaries

A *feature* is a variable that encodes information about an object. We speak of *binary* f_{binary}, *categorical* $f_{categorical}$, and *continuous* features $f_{continuous}$ depending on their domain, i.e., $f_{binary} \in \{0, 1\}$, $f_{integer} \in \mathbb{N}$, and $f_{continuous} \in \mathbb{R}$. A *feature vector* is a vector of features. An *instance* is a pair that consists of a feature vector and a value representing the *class*. A class can take continuous or discrete values. A *dataset*, or simply *data*, is a set of instances. While features within a vector may have different domains, the *i-th* feature of each feature vector of the dataset shares the same domain. The assumption is that the features describe certain characteristics about the objects, and the *i-th* feature of each feature vector refers to the same characteristic of interest.

The process of *learning* seeks to compute a *learning function* that performs *classification*, i.e., maps feature vectors to classes. The target learning function is restricted to a particular form, e.g., the form of a decision tree (see further), and the goal is to compute a function

that minimises or maximises the target metric for a given dataset. If the domain of the classes of the dataset is discrete, we speak of a *classification* problem, and otherwise of a *regression* problem for continuous classes.

Decision trees are binary trees from computer science. We call leaf and non-leaf nodes *classification* and *predicate* nodes, respectively. Each predicate node is given a predicate that maps feature vectors to a Boolean value, i.e., $\{0, 1\}$. The left and right edges of a predicate node are associated with the values zero and one, respectively. Each classification node is assigned a fixed class.

A decision tree is a learning function that performs classification according to the following recursive procedure. Given a feature vector, it starts by considering the root node. If the considered node is a classification node, its class determines the class of the feature vector and the procedure terminates. Otherwise, the node is a predicate node, and the left child node will be considered next if the predicate of the node evaluates to zero, and otherwise the right child node is selected. The process recurses until a class is determined.

The *(feature) depth* of a decision tree is the maximum number of feature nodes any instance may encounter during classification. The *size* of a decision tree is the number of feature nodes. It follows that the maximum size of a decision tree with depth d is $2^d - 1$. We note that in the literature, in some cases, the size is defined as the total number of nodes in the tree. These definitions are equivalent and can be used interchangeably, as a tree with n predicate nodes has $n + 1$ classification nodes.

In practice, the predicates take a special form. For *single-variate* or *axis-aligned* decision trees, which are the focus of this work, predicates only consider a single feature and typically test whether it exceeds a threshold value. We refer to these nodes are *feature nodes*, as the predicate depends solely on one feature. Furthermore, the predicates are chosen based on the dataset. Generalisations of decision trees are straight-forward: *multi-variate* versions use predicates that operate on more than one feature, and predicates can be substituted by functions whose co-domains are of size n, in which case the decision tree is an n-ary tree with an analogous definition. These generalisations are mentioned for completeness and are not further discussed.

We use special notation for *binary datasets*, where the domain of features and classes is Boolean. Given a feature vector $\mathbf{fv}$, we write $f_i \in \mathbf{fv}$ and $\overline{f_i} \in \mathbf{fv}$ if the *i-th* feature has value one and zero, respectively. The value one indicates the feature is present in the feature vector, and otherwise it is not present. Features f_i and $\overline{f_i}$ are referred to as *positive* and *negative* features, respectively. We limit the predicates to only output the value of a particular feature in the feature vector and simply write f_i and $\overline{f_i}$ for the predicates. The binary dataset $\mathcal{D}$ is partitioned into a positive and negative class of instances based on the classes, i.e., $\mathcal{D} = \mathcal{D}^+ \cup \mathcal{D}^-$. We consider the partitions as sets of feature vectors since their class is clear from context, and write $\mathcal{D}(f)$ as the set of instances from $\mathcal{D}$ that contain feature f, and analogously for multiple features, e.g., $\mathcal{D}(f_1, f_2)$ are the set of instances that contain both f_1 and f_2. The *misclassification score* of a decision tree on data is the number of instances for which classification produces the incorrect class considering the data as ground truth.

3. Literature Review

Historically the most popular techniques for decision tree learning were based on heuristics due to their effectiveness and scalability. Examples of these algorithms include CART, originally proposed by Breiman et al. (1984), and C4.5 by Quinlan (1993). These algorithms start with a single node, and iteratively expand the tree based on metrics such as information gain and Gini coefficients, and possibly post-process the obtained decision trees to prune branches in an effort to reduce overfitting. While there is a vast literature on heuristic algorithms for decision trees, in this work we are primarily concerned with *optimal decision trees*, and hence direct further discussion to such settings.

Bertsimas and Shioda (2007) presented a mixed-integer programming approach for optimal decisions that worked well on smaller datasets. Mixed-integer programming formulations with better performance were given by Bertsimas and Dunn (2017) and Verwer and Zhang (2017). These methods encode the optimal decision tree by fixing the tree depth in advance, creating variables to represent the predicates for each node, and adding constraints to enforce the decision tree structure. These approaches were later improved by *BinOPT* (Verwer and Zhang (2019)), a *binary linear programming* formulation, that took advantage of implicitly binarising data to reduce the number of variables and constraints required to encode the problem. Aghaei et al. (2019) used a mixed-integer programming formulation for optimal decision trees that supported *fairness* metrics. The authors argued that using machine learning in socially sensitive contexts may perpetuate discrimination if no special measures are taken into account. They propose fairness metrics and incorporate them in a mixed-integer programming formulation.

An encoding of decision trees using propositional logic (SAT) has been devised by Narodytska et al. (2018). In this line of work, the aim is to construct the smallest tree in terms of the total number of nodes that *perfectly* describes the given dataset, i.e., leads to zero misclassifications on the training data. An initial perfect decision tree is constructed using a heuristic method, after which a series of SAT-solver calls are made, each time posing the problem of computing a perfect tree with one less node. The SAT approach of Avellaneda (2020) simplifies the encoding by fixing the depth of the tree and employs an incremental approach where instances are gradually added to the formulation rather than being considered completely from the start.

Nijssen and Fromont (2007) introduced a framework named *DL8* for optimal decision trees that could support a wide range of constraints. They took advantage that the left and right subtree of a given node can be optimised independently, introduced a caching technique to save subtrees computed during the algorithm in order to reuse them at a later stage, and combined these with ideas from the pattern mining literature to compute optimal decision trees. DL8 laid an important foundation for optimal decision tree algorithms that follow.

Verhaeghe et al. (2019) approached the optimal classification tree problem by minimising the misclassifications using constraint programming. The independence of the left and right subtrees from Nijssen and Fromont (2007) was captured in an AND-OR search framework. Upper bounding on the number of misclassifications was used to prune parts of the search space and their algorithm incorporated an itemset mining technique to speed-

up the computation of instances per node and used a caching technique similar to DL8 (Nijssen and Fromont (2007)),

Hu et al. (2019) presented an algorithm that computes the optimal decision tree by considering a balance between misclassifications and number of nodes. They apply exhaustive search, caching, and lower bounding of the misclassifications based on the cost of adding a new node to the decision tree. Compared to other recent optimal decision tree algorithms, the method relies on the number of nodes playing an important role in the metric of optimality and a limited number of binary features, e.g., the authors experimented with datasets with up to twelve binary features.

Aglin et al. (2020) developed *DL8.5* by combining and refining the ideas from *DL8* and the constraint programming approach. Their main addition was an upper bounding technique, which limited the upper misclassification value of a child node once the optimal subtree was computed for its sibling, and a lowering bound technique, where the algorithm stored information not only about computed optimal subtrees but also pruned subtrees to provide a lower bound on the misclassifications of a subtree. This led to an algorithm that outperformed previous approaches by a notable margin.

Exploiting properties specific to the decision tree learning problem proved to be valuable in improving algorithmic performance in previous work. In particular, search and pruning techniques, caching computation for later reuse, and the techniques that take advantage of the decision tree structure all lead to notable gains in performance. These are the main reasons for the success of specialised methods over generic frameworks, such as integer programming and SAT. As there is a significant overlap of ideas and techniques used in related work, we discuss these in more detail in Section 4.1 when presenting the high-level view of our framework.

Lastly, we refer the readers to a curated list of decision tree papers by Benedek Rozemberczki: `https://github.com/benedekrozemberczki/awesome-decision-tree-papers`.

4. MurTree: Our Framework for Optimal Classification Trees

Our framework computes optimal classification trees by exhaustive search. The search space is exponentially large, but special measures are taken to efficiently iterate through solutions, exploit the overlap between solutions, and avoid computing suboptimal decision trees.

We give the main idea of the algorithm, then provide the full pseudo code, and follow up with individual subsections where we present each individual technique in greater detail.

For the sake of clarity, the remaining text focusses on optimal classification trees that minimise the number of misclassified instances for binary datasets and binary classification. Extending the framework for general settings, such as continuous and categorical data, is discussed in Section 4.9.

4.1 High-Level Idea

We note two important properties of decision trees:

Property 1 *(Independence) Given a dataset $\mathcal{D}$, a feature node partitions the dataset $\mathcal{D}$ into its left and right subtree, such that $\mathcal{D}_{left} \cap \mathcal{D}_{right} = \emptyset$ and $\mathcal{D} = \mathcal{D}_{left} \cup \mathcal{D}_{right}$.*

Property 2 *(Overlap) Given a classification node, a set of features encountered on the path from the root node to the classification node, and an instance, the order in which the features are used to evaluate the instance does not change the classification result.*

Both properties follow directly from the definition of decision trees and are emphasised as they play a major role in designing decision tree algorithms. Property 1 allows computing the misclassification score of the tree as the sum of the misclassification scores of its left and right subtree, and as will be discussed, once a feature node is selected, the left and right subtrees can be optimised independently of each other. Property 2 shows there is an overlap between decision trees that share the same features, which is taken advantage of by caching techniques (see Section 4.6 for more details).

The dynamic programming formulation of optimal classification trees given in Eq. 1 provides a high-level summary of our framework. The input parameters consist of a binary dataset $\mathcal{D}$ with features $\mathcal{F}$, an upper bound on depth d, and an upper bound on the number of feature nodes n. The output is the minimum number of misclassifications possible on the data given the input decision tree characteristics. The key observations are given by Properties 1 and 2. The two observations, independence and overlap, when combined reveal the dynamic programming structure of decision trees.

$$T(\mathcal{D}, d, n) = \begin{cases} T(\mathcal{D}, d, 2^d - 1) & n > 2^d - 1 \\ \min\{|\mathcal{D}^+|, |\mathcal{D}^-|\} & n = 0 \\ \min\{T(\mathcal{D}(\overline{f}), d-1, n-i-1) & \textit{general case} \\ \quad + T(\mathcal{D}(f), d-1, i) : f \in \mathcal{F}, i \in [0, n-1]\} & \end{cases} \tag{1}$$

The first case in Eq. 1 places a natural limit on the number of feature nodes given the depth. The second case defines the misclassification score for classification nodes. The general case states that computing the optimal misclassification score amounts to examining all possible feature splits and ways to distribute the feature node count to the left and right child of the root node. For each combination of a selected feature and node count distribution to its children, the optimal misclassification is computed recursively as the sum of the optimal misclassifications of its children. The formulation is exponential in the depth, feature node limit, and number of features, but with special care, as presented in the subsequent sections, it is possible to compute practically relevant optimal classification trees within a reasonable time.

Eq. 1 serves as the core foundation of our framework. In contrast to related work, we take advantage of the structure of decision trees to allow imposing a limit on the number of nodes as presented in Eq. 1. Previous approaches either place no constraint on the number of nodes apart from the depth (Nijssen and Fromont (2007); Aglin et al. (2020)), limit the number of nodes by penalising the objective function for each node in the tree (Hu et al. (2019)), or allow constraints on the number of nodes but do not make use of decision tree properties (Bertsimas and Dunn (2017); Narodytska et al. (2018); Verwer and Zhang (2017, 2019); Avellaneda (2020)). The last point is particularly important as the ability to exploit optimal decision tree properties is essential for achieving the best performance.

Simpler and/or modified forms of Eq. 1 were used in some previous work under different terminology. The AND-OR search method (Verhaeghe et al. (2019)), pattern mining

approach (Nijssen and Fromont (2007); Aglin et al. (2020)), and the search by Hu et al. (2019) use the independence property of the left and right subtree (Property 1). Those approaches save computed optimal subtrees (Property 2), which corresponds to *memoisation* as an integral part of dynamic programming (Section 4.6). Framing the problem as a dynamic program dates from the 1970s (e.g., Garey (1972)), but the description in works afterwards deviated as new techniques were introduced. We present the problem back in its original dynamic programming format and together with our node limitation addition, unite and generalise previous approaches using conventional algorithmic notation.

A key component of our framework is a specialised algorithm for computing decision trees of depth at most two. It takes advantage of the specific decision tree structure by performing a precomputation on the data, which allows it to compute the optimal decision tree without explicitly referring to the data. This offers a significantly lower computational complexity compared to the generic case of Eq. 1, but is applicable in practice only to decision trees of depth two. Thus, rather than following Eq. 1 until the base case, we stop the recursion once a tree of depth two is required and invoke the specialised method.

A defining characteristic of search algorithms are pruning techniques, which detect areas of the search that may be discarded without losing optimality. In the case of decision trees, subtrees may be pruned based on the lower or upper bound of the number of misclassifications of the given subtrees. If the bound shows that the misclassifications of a currently considered subtree will result in a high value, the subtree can be pruned, effectively reducing the search space. The challenge when designing bounding techniques is to find the correct balance between pruning power and the computational time required by the technique.

We introduce a novel similarity-based lower bounding technique (Section 4.5) that derives a bound based on the similarity of the previously considered subtrees. We use our lower bounding method in combination with the previous lower bounding approach introduced in DL8.5 (Aglin et al. (2020)), which we describe in the following text. Given a parent node, once the optimal subtree is computed for one of the children, an upper bound can be posed on the other child subtree based on the best decision tree known for the parent node and the number of misclassifications of the optimal child subtree. If a subtree fails to produce a solution within the posed upper bound, the upper bound is effectively a lower bound that can be used once the same subtree is encountered again in the search. Our algorithms uses a refinement of the described lower bound, which additionally takes into account all lower bounds of the children of the parent node (Section 4.6.1). Hu et al. (2019) uses a bound for an objective function that balances the accuracy (misclassifications) and number of nodes in the tree. If α is the penalty in terms of misclassifications for adding a node to the decision tree, then α also serves as a lower bound for each subtree (otherwise it is not worth introducing a node). We do not incorporate this last bound explicitly in our framework, but instead compute trees with such objective functions by solving a series of (overlapping) trees that optimise only the misclassification score (Section 4.9.3).

The remaining part of the paper describes our techniques in more detail.

4.2 Main Loop of the Framework

Algorithm 1 summarises our framework. As discussed in the previous section, it can be seen as an instantiation of Eq. 1 with additional techniques to speed-up the computation.

The algorithm takes as input a dataset $\mathcal{D}$ consisting of positive $\mathcal{D}^+$ and negative $\mathcal{D}^-$ instances, the maximum depth and size (number of feature nodes) of the decision tree, and an upper bound that represents a limit on the number of misclassifications before the tree is considered infeasible. The output is an optimal classification tree respecting the input constraints on the depth, size, and upper bound, or a flag indicating that no such tree exists, i.e., the problem is *infeasible*. The latter occurs as a result of recursive calls (see further), which pose an upper bound that is necessary to ensure the decision tree has a lower misclassification value than the best tree found so far in the search.

The upper bound is initially set to the misclassification score of a single classification node for the data and is updated throughout the execution. Note that at the start of the algorithm a tighter upper bound could be computed by using a heuristic algorithm.

The base case of Eq. 1 is initially tested and a classification node is returned if no feature nodes are allowed. In addition, subtrees that are at their lower bound misclassification values are already optimal and are returned immediately.

After the initial tests, the algorithm attempts to prune the current tree based on two lower bounds: our novel similarity-based approach (Section 4.5), and our generalisation of the cache-based lower bounding introduced in DL8.5 (Aglin et al. (2020)) but now extended to take into account the number of nodes in the tree.

Assuming pruning did not take place, if the current subtree has already been computed as part of a previous recursive call, the solution is retrieved from the cache and the current call terminates. Caching subtrees for trees where the depth is constrained dates from DL8 (Nijssen and Fromont (2007)). In our work, the algorithm caches with respect to the depth and number of node constraints.

A key aspect of our framework is that trees of depth at most two are computed using a specialised procedure (Section 4.3). It solves the optimal decision tree problem in a sense as a unit operation and ignores the upper bound. The result is stored in the cache for future computation regardless of the feasibility of tree with respect to the upper bound, but the upper bound determines if the obtained tree is considered feasible.

If none of the above criteria is met, the algorithm reaches the general case from Eq. 1, where the search space is exhaustively explored through a series of overlapping recursions (Algorithm 2).

Recall that the size of tree, i.e., the number of feature nodes, is given as input. One node is allocated as the root, and the remaining node budget is split among its children. For each feature, the algorithm considers all possible combinations of distributing the remaining node budget to its left and right subtrees. Note that determining the maximum size of thfe left subtree immediately fixes the maximum size of the right subset, and that special care needs to be taken to not allocate a size to a subtree that is greater than it may support with respect to its depth.

For a chosen tree configuration (the feature of the subtree root and the size of its subtrees), the algorithm determines which subtree to recurse on first. Previous work in DL8 and DL8.5 fixed the order by exploring the left before the right subtree. In our framework, we introduce a dynamic strategy that prioritises the subtree with the largest gap between its lower and upper bound (Section 4.7). The intuition is that this subtree is more probable to have a higher misclassification score, which in turn increases the likelihood of pruning the other sibling.

The algorithm then solves the subtrees in the chosen order. If the first subtree is infeasible, this implies that the lower bound of the subtree is one greater than the given upper bound. The information is stored in the cache in case the bound is needed in one of the other recursive calls. This bound was introduced in DL8.5 Aglin et al. (2020) and we provide a further refinement by into account pairwise sum of the lower bounds of the children (Section 4.6.1). Recall that the misclassification score of the root is the sum of the misclassifications of its children, and therefore the second subtree can be discarded if its sibling already led to an infeasible tree.

If both recursive calls successfully terminated, the obtained decision tree is recorded as the best tree found so far and the solution is stored in the cache. In our framework, as soon as a new globally optimal decision tree is encountered, it is identified as such. This leads to fully *anytime behaviour*, i.e., the execution can be stopped at any given point in time to return its current best solution. In the previous work of DL8.5, for instance, a globally improving solution was only detected at the root node of the complete decision tree.

Once all the recursive calls have been completed, the search space of the subtree has been exhaustively explored. The cache is updated with respect to the best locally found subtree: either the subtree is stored in the cache as an optimal subtree, or its lower bound is updated in case no feasible subtree was found. In this manner, all possible decision trees are explored and a tree with minimum misclassification score is returned.

The dynamic programming aspect can be seen as the method divides the main problem into smaller overlapping subproblems, owing to Properties 1 and 2. Search is used to prune the search space, saving computation time, and drives the algorithm towards the specialised algorithm, which efficiently computes optimal subtrees of depth at most two. The last point is a key component in reducing the overall runtime compared to previous approaches, as discussed in Section 4.3.

Lastly, we note two points not included in the pseudo-code for simplicity. Note the following definition and proposition:

Definition 1 *(Degenerate Decision Trees) A decision tree is degenerate if it contains at least one classification node that does not classify any training instance.*

Proposition 2 *(Pruning Degenerate Trees) Given a degenerate decision tree with n feature nodes and misclassification score s on the training data, there exists at least one other decision tree with $n' < n$ feature nodes and misclassification score $s' \leq s$.*

Degenerate trees may occur during the algorithm when splitting on a nondiscriminative feature, such that one subtree contains no training instances, i.e., $|\mathcal{D}| = 0$. Due to Proposition 2, we deem these trees infeasible and prune them as soon as they are detected.

The second point is that the initial best subtree is set to a classification node if allowed by the upper bound, rather than an infeasible tree as given in Algorithm 1, which may trigger a global update of the best solution.

This concludes the description of the main loop of our framework. Before proceeding with detailing each component of our algorithm, we reiterate the differences between our approach and DL8.5 (Aglin et al. (2020)) in light of the technical description given above.

Comparison with DL8.5 (Aglin et al. (2020)). By virtue of taking into account the structure of decision trees, Algorithm 1 shares a similar layout as in DL8.5, but there are notable

differences that result in orders of magnitude speed-ups. The differences can be summarised as follows: 1) we allow constraining the size of tree in addition to the depth, which is important in obtaining the smallest optimal decision, e.g., to improve interpretability or learn trees that generalise on unseen instances (Section 5.4), 2) our specialised algorithm (Section 4.3) is substantially more efficient at computing trees with depth two when compared to the general algorithm in Algorithm 1 or DL8.5, 3) we propose a new lower bound based on the similiarity with previously computed subtrees to further prune the search space (Section 4.5) and refine the previous lower bound (4.6.1), 4) our cache policy (Section 4.6) is extended to support the size of the tree constraint and allows for *incremental solving*, allowing reusing computation when solving trees with increasing depth and size, e.g., during hyper-parameter tuning, 5) we dynamically determine which subtree to explore first based on pruning potential (Section 4.7), rather than use a static stategy, and 6) our framework immediately updates the best global solution as soon as it is computed rather than only at the root node.

4.3 Specialised Algorithm for Trees of Depth Two

An essential part of our framework is a specialised method for computing optimal decision trees of depth two. The procedure is repeatedly called in our framework, i.e., each time a tree of at most depth two needs to be optimally solved. In the following, we present an algorithm that achieves lower complexity than the general algorithm (Eq. 1 and Prop. 3) when considering trees with depth two.

Prior to presenting our specialised algorithm, we discuss the complexity of computing decision trees of depth two using Eq. 1 as the baseline.

Proposition 3 *Computing the optimal classification tree of depth two using Eq. 1 can be done in* $\mathcal{O}(|\mathcal{D}| \cdot |\mathcal{F}|^2)$ *time.*

Assume that splitting the data based on a feature node is done in $\mathcal{O}(|\mathcal{D}|)$ time. Eq. 1 considers $|\mathcal{F}|$ splits for root and for each feature performs $2 \cdot |\mathcal{F}|$ splits for its children. This results in $2 \cdot |\mathcal{F}|^2$ splits and an overall runtime of $\mathcal{O}(|\mathcal{D}| \cdot |\mathcal{F}|^2)$, proving Proposition 3. In practice, partitioning the dataset based on a feature can be sped-up using bitvector operations and caching subproblems (Aglin et al. (2020); Verhaeghe et al. (2019); Hu et al. (2019)), but the complexity remains as this only impacts the hidden constant in the big-O.

In the following, we present an algorithm with lower complexity and additional practical improvements which, when combined, reduce the runtime of computing the optimal classification tree of depth two by orders of magnitudes.

4.4 Algorithm Description

Algorithm 3 provides a summary. The input is a dataset $\mathcal{D}$ and the output is the optimal classification tree of depth two with three feature nodes that minimises the number of misclassified instances.

The specialised procedure computes the optimal decision tree in two phases. In the first step, it computes frequency counts for each pair of features, i.e., the number of instances in which both features are present. In the second step, it exploits the frequency counts to efficiently enumerate decision trees without needing to explicitly refer to the data. This

Algorithm 1: $MurTree(\mathcal{D}, depth, size, UB)$, a dynamic programming and search algorithm for computing optimal classification trees

input: Dataset $\mathcal{D} = \mathcal{D}^+ \cup \mathcal{D}^-$, $size \in \mathbb{N}$, $depth \in \mathbb{N}$, upper bound on the misclassifications $UB \in \mathbb{N}$

output: Optimal classification tree within the input *size* and *depth* that minimises the misclassification score on $\mathcal{D}$

begin
  // Base case (Eq. 1): no feature nodes are possible or the node is already at its lower bound
  if $depth = 0 \vee size = 0 \vee LB(\mathcal{D}, depth, size) = classification\_node(\mathcal{D})$ **then**
    return $classification\_node(\mathcal{D})$
  // Prune using similiarity-based lower bounding (Section 4.5) if possible
  if $SimiliarityLowerBound(\mathcal{D}, depth, size) > UB$ **then**
    return *infeasible*
  // Prune using cache-based pruning (Section 4.6.2) if possible
  if $CachedLowerBound(\mathcal{D}, depth, size) > UB$ **then**
    return *infeasible*
  // Use cached subtrees if possible (Section 4.7)
  if *subtree* $(\mathcal{D}, depth, size)$ *has already been computed* **then**
    return $GetCachedSubtree(\mathcal{D}, size, depth)$
  // Use the specialised algorithm from Section 4.3 if possible
  if $depth \leq 2$ **then**
    $best\_subtree \leftarrow SpecialisedAlgorithm(\mathcal{D}, size, depth)$
    // Store the subtree regardless of the UB since it is optimal (Section 4.6.1)
    $UpdateCache(\mathcal{D}, depth, size, best\_subtree)$
    if $score(best\_subtree) > UB$ **then**
      return *infeasible*
    else
      return $best\_subtree$
  // General case (Eq. 1): exhaustively search using Algorithm 2
  $best\_tree \leftarrow MurTree.GeneralCase(\mathcal{D}, depth, size, UB)$
  return $best\_tree$

Algorithm 2: $MurTree.GeneralCase(\mathcal{D}, depth, size, UB)$, the general case of Eq. 1 in Algorithm 1

begin
  // General case (Eq. 1): exhaustively explore the search space
  $best\_tree \leftarrow infeasible$
  // RLB is the lower bound using Eq. 15
  $RLB \leftarrow \infty$
  for *feature* $f \in \mathcal{F}$ **do**
    $max\_size\_subtree \leftarrow \min\{2^{(depth-1)} - 1, size - 1\}$
    $min\_size\_subtree \leftarrow (size - 1 - max\_size\_subtree)$
    for $left\_size \in [min\_size\_subtree, max\_size\_subtree]$ **do**
      $right\_size \leftarrow (size - 1 - left\_size)$
      // Dynamic post-order: process left subtree first (Section 4.7)
      if $LowerBound(\mathcal{D}(\overline{f}), depth - 1, left\_size) <$
       $LowerBound(\mathcal{D}(f), depth - 1, right\_size)$ **then**
        $left\_subtree \leftarrow MurTree(\mathcal{D}(\overline{f}), left\_size, depth - 1, UB)$
        // No need to compute the other subtree if this one failed
        if *left_subtree is infeasible* **then**
          $RLB \leftarrow \min\{RLB, LowerBound(\mathcal{D}(\overline{f}), depth - 1, left\_size) +$
           $LowerBound(\mathcal{D}(f), depth - 1, right\_size)\}$
          continue
        $right\_size \leftarrow (size - 1 - left\_size)$
        $right\_subtree \leftarrow$
         $MurTree(\mathcal{D}(f), right\_size, depth - 1, UB - score(left\_subtree))$
        // If both children are feasible, update the globally and locally best solution, the cache (Section 4.6), and the upper bound
        if *right_subtree is feasible* **then**
          $best\_subtree \leftarrow DecisionTree(f, left\_subtree, right\_subtree)$
          $UpdateGlobalSolution(best\_subtree, depth, size)$
          $UpdateCache(\mathcal{D}, depth, size, best\_subtree)$
           $UB \leftarrow score(best\_subtree) - 1$
        else
          $RLB \leftarrow \min\{RLB, LowerBound(\mathcal{D}(\overline{f}), depth - 1, left\_size) +$
           $LowerBound(\mathcal{D}(f), depth - 1, right\_size)\}$
      else
        // Dynamic post-order: process right subtree first (Section 4.7)
        Process right subtree analogously as above
  // Cache the optimal solution or record the lower bound (Section 4.6.1)
  $UpdateCache(\mathcal{D}, depth, size, UB, RLB, best\_subtree)$
  return $best\_tree$

provides a substantial speed-up compared to iterating through features and splitting data as given in the dynamic programming formulation (Eq. 1) for decision trees of depth two. We now discuss each phase in more detail and present a technique to incrementally compute the frequency counts.

Algorithm 3: Specialised algorithm for computing optimal classification trees of depth two with three nodes

input: Binary dataset $\mathcal{D} = \mathcal{D}^+ \cup \mathcal{D}^-$
output: Optimal classification tree of depth two with three feature nodes that minimises the misclassification score on $\mathcal{D}$

begin
  $\forall f_i : FQ^+(f_i) \leftarrow 0 \wedge FQ^-(f_i) \leftarrow 0$
  $\forall f_i, f_j, i < j : FQ^+(f_i, f_j) \leftarrow 0 \wedge FQ^-(f_i, f_j) \leftarrow 0$
  /* Step 1: construct the frequency counter of positive features */
  for fv $\in \mathcal{D}^+$ **do**
    for $f_i \in$ **fv do**
      increment $FQ^+(f_i)$
      for $f_j \in$ **fv** $s.t.\ i < j$ **do**
        increment $FQ^+(f_i, f_j)$
  FQ^- is computed as above using $\mathcal{D}^-$
  /* Step 2: construct the optimal decision tree based on the frequency counters FQ^+ and FQ^- */
  /* Compute the best left and right subtrees for each feature */
  for $f_i \in \mathcal{F}$ **do**
    for $f_j \in \mathcal{F}$ $s.t.\ i \neq j$ **do**
      $CS(\overline{f_i}, f_j) \leftarrow \min\{FQ^+(\overline{f_i}, f_j), FQ^-(\overline{f_i}, f_j)\}$
      $CS(\overline{f_i}, \overline{f_j}) \leftarrow \min\{FQ^+(\overline{f_i}, \overline{f_j}), FQ^-(\overline{f_i}, \overline{f_j})\}$
      /* Compute branch with f_i as root and f_j as left child */
      $MS_{left}(f_i, f_j) \leftarrow CS(\overline{f_i}, \overline{f_j}) + CS(\overline{f_i}, f_j)$
      if $BestLeftSubtree(f_i).misclassification > MS_{left}(f_i, f_j)$ **then**
        $BestLeftSubtree(f_i).misclassification \leftarrow MS_{left}(f_i, f_j)$
        $BestLeftSubtree(f_i).feature \leftarrow f_j$
      The best right subtree with f_i as the root and f_j as the right child is computed analogously as above
  /* Compute the best tree by taking the feature with the minimum sum of misclassification of its children */
  $best\_tree \leftarrow argmin_{f_i \in \mathcal{F}}\{BestLeftSubtree(f_i).misclassification + BestRightSubtree(f_i).misclassification\}$
  return $best\_tree$

4.4.1 Phase One: Frequency counting (Algorithm 3, Lines 2-9)

Let $FQ^+(f_i)$ and $FQ^+(f_i, f_j)$ denote the frequency counts in the positive instances for a single feature and a pair of features, respectively. The functions $FQ^-(f_i)$ and $FQ^-(f_i, f_j)$ are defined analogously for the negative instances.

A key observation is that based on $FQ(f_i)$ and $FQ(f_i, f_j)$, we may compute $FQ(\overline{f_i})$, $FQ(f_i, \overline{f_j})$, $FQ(\overline{f_i}, f_j)$, and $FQ(\overline{f_i}, \overline{f_j})$. This is done as follows:

$$FQ^+(\overline{f_i}) = |\mathcal{D}^+| - FQ^+(f_i) \tag{2}$$

$$FQ^+(f_i, \overline{f_j}) = FQ^+(f_i) - FQ^+(f_i, f_j) \tag{3}$$

$$FQ^+(\overline{f_i}, f_j) = FQ^+(f_j) - FQ^+(f_i, f_j) \tag{4}$$

$$FQ^+(\overline{f_i}, \overline{f_j}) = |\mathcal{D}^+| - FQ^+(f_i) - FQ^+(f_j) - FQ^+(f_i, f_j) \tag{5}$$

The equations make use of the fact that the features are binary. For example, Eq. 2 states that if the total number of positive instances is $|\mathcal{D}^+|$ and we computed the frequency count $FQ^+(f_i)$, then the frequency count $FQ^+(\overline{f_i})$ is the number of instances in which f_i does not appear, i.e., the difference between $|\mathcal{D}^+|$ and $FQ^+(f_i)$. Similar reasoning is applied to the other equations and computing the frequency count FQ^- is analogous.

The following proposition summarises the runtime of computing $FQ^+(f_i, f_j)$.

Proposition 4 *(Computational Complexity of Phase One) Let m_+ denote the maximum number of features in any single positive instance. Frequency counts $FQ^+(f_i, f_j)$ can be computed in $\mathcal{O}(|D^+| \cdot m_+^2)$ time with $\mathcal{O}(\mathcal{F}^2)$ memory.*

An efficient way of computing the frequency counts is to represent the feature vector as a *sparse* vector, and iterate through each instance in the dataset and increase a counter for each individual feature and each pair of features. This leads to the proposed complexity result. The additional memory is required to store the frequency counters, allowing to query a frequency count as a constant time operation. Note that the pairwise frequency count is symmetric, i.e., $FQ^+(f_i, f_j) = FQ^+(f_j, f_i)$, which requires only to consider f_i and f_j in the frequency count for $i < j$. This results in a smaller hidden constant in the big-O notation.

4.4.2 Phase Two: Optimal tree computation (Algorithm 3, Lines 10-19)

Recall that a classification node is assigned the positive class if the number of positive instances exceeds the number of negative instances, otherwise the node class is negative. Let $CS(f_i, f_j)$ be the classification score for a classification node with all instances of $\mathcal{D}$ containing both features f_i and f_j. The classification score is then computed as follows.

$$CS(f_i, f_j) = \min\left\{FQ^+(f_i, f_j), FQ^-(f_i, f_j)\right\} \tag{6}$$

Given a decision tree with depth two, a root node with feature f_{root}, a left and right child node with features f_{left} and f_{right}, we may compute the misclassification score in

constant time assuming the frequency counts are available. Let MS_{left} and MS_{right} denote the misclassification scores of the left or right subtree. The computations are as follows.

$$MS_{left}(f_{root}, f_{left}) = CS(\overline{f_{root}}, \overline{f_{left}}) + CS(\overline{f_{root}}, f_{left}) \tag{7}$$

$$MS_{right}(f_{root}, f_{right}) = CS(f_{root}, \overline{f_{right}}) + CS(f_{root}, f_{right}) \tag{8}$$

The total misclassification score of the tree is the sum of misclassifications of its children. As the number of misclassification can be computed solely based on the frequency counts, we may conclude the computational complexity.

Proposition 5 *(Computational Complexity of Phase Two) Given the frequency counts FQ^+ and FQ^-, the optimal subtree tree can be computed in $\mathcal{O}(|F|^2)$ time with $\mathcal{O}(|F|)$ memory.*

It follows from Property 1 that given a root node with feature f_{root}, the left and right subtrees can be optimised independently. Therefore, it is sufficient to compute for each feature its best left and right subtrees, and take the feature with the minimum sum of its child misclassifications. To compute the best left and right feature for each feature, the algorithm maintains information about the best left and right child for each feature found so far, leading to the memory requirement from Proposition 5. The best features are initially arbitrarily chosen. Recall that from Property 1 it follows that the left and right subtree can be optimised independently:

$$\min_{f_{left}, f_{right} \in \mathcal{F}} MS(f_{root}, f_{left}, f_{right}) = \min_{f_{left} \in \mathcal{F}} MS_{left}(f_{root}, f_{left}) + \min_{f_{right} \in \mathcal{F}} MS_{right}(f_{root}, f_{right})$$

Therefore, rather than considering triplets of features $(f_{root}, f_{left}, f_{right})$, it iterates through each pair of features (f_{root}, f_{child}), computes the misclassification values of the left subtree using Eq. 7, updates the best left child for feature f_{root}, and performs the same procedure for the right child. After iterating through all pairs of features, the best left and right subtree is known for each feature, leading to the proposed complexity. The optimal decision tree can then be computed by finding the feature with minimum misclassification cost of its combined left and right misclassification.

After discussing each individual phase, we may conclude the overall complexity:

Proposition 6 *(Computational Complexity of Depth-2 Decision Trees) Let m be the upper limit on the number of features in any single positive and negative instance. The number of operations required to computing an optimal decision tree is $\mathcal{O}(|D| \cdot m^2 + |\mathcal{F}|^2)$ using $\mathcal{O}(\mathcal{F}^2)$ auxiliary memory.*

The result follows by combining Propositions 4 and 5. The obtained runtime is substantially lower at the expense of using additional memory compared to the dynamic programming formulation (Eq. 1) outlined in Proposition 3. Note that instances with binary features are naturally sparse. If the majority of instances contain more than half of the features, then as a preprocessing step all feature values may be inverted to achieve sparsity

without loss of generality. The advantage of our approach is exemplified with lower sparsity ratios, i.e., small m values.

There are several additional points to note, which are not shown in Algorithm 3 to keep the pseudo-code succinct.

The above discussion assumed the feature node limit was set to three. The algorithm can be modified for the case of two feature nodes, keeping the same complexity, while in the case with only one feature node the pairwise computations are no longer necessary leading to $\mathcal{O}(|D| \cdot m + |F|)$ complexity. Similarly, the algorithm is implemented to lexicographically minimise the misclassification score and then the size of the tree.

To improve the performance in practice, the algorithm iterates through pairs of features (f_i, f_j) such that $i < j$. After updating the current best left and right subtree feature using f_i as the root and f_j as the child, the same computation is done using f_j as the root and f_i as the child. Compared to the pseudo-code in Algorithm 3, this cuts the number of iterations by half, but each iteration does twice as much work, which results in a speed-up in practice. Moreover, rather than computing the best tree in a separate loop after computing the best left and right subtrees for each feature, this is done on the fly by keeping track of the best subtree encountered so far during the algorithm.

Specialised algorithm for decision trees of depth three. We considered computing decision trees with depth three using a similar idea. Even though this results in a better big-O complexity for trees of depth three, albeit requiring $\mathcal{O}(\mathcal{F}^3)$ memory, our preliminary results did not indicate practical benefits. Including additional low-level optimisation might improve the results, but for the time being we leave this as an open question.

4.4.3 Incremental Computation

The specialised method for computing decision trees of depth two is repeatedly called in the framework. For each call, the algorithm is given a different dataset that is a result of applying a split in one of the nodes in the tree. The key observation is that datasets which differ only in a small number of instances result in *similar* frequency counts. The idea is to exploit this by only updating the necessary difference rather than recomputing the frequency counts from scratch.

The key point is to view the previous dataset $\mathcal{D}_{old}$ and the new dataset $\mathcal{D}_{new}$ in terms of their intersection and differences.

Observation 1 *Given two datasets $\mathcal{D}_{new}$ and $\mathcal{D}_{old}$, let their difference be denoted as $\mathcal{D}_{in} = \mathcal{D}_{new} \setminus \mathcal{D}_{old}$ and $\mathcal{D}_{out} = \mathcal{D}_{old} \setminus \mathcal{D}_{new}$ and their intersection as $\mathcal{D}_{same} = \mathcal{D}_{new} \cap \mathcal{D}_{old}$. We may express the datasets as $\mathcal{D}_{new} = \mathcal{D}_{in} \cup \mathcal{D}_{same}$ and $\mathcal{D}_{old} = \mathcal{D}_{out} \cup \mathcal{D}_{same}$*

We first note that set operations can be done efficiently for datasets.

Proposition 7 *(Computational Complexity of Set Operations on Datasets) Given a dataset $\mathcal{D}$ and two of its subsets $\mathcal{D}_{new} \subseteq \mathcal{D}$ and $\mathcal{D}_{old} \subseteq \mathcal{D}$, the sets $\mathcal{D}_{in} = \mathcal{D}_{new} - \mathcal{D}_{old}$ and $\mathcal{D}_{out} = \mathcal{D}_{old} - \mathcal{D}_{new}$ can be computed in $\mathcal{O}(|\mathcal{D}_{new}| + |\mathcal{D}_{old}|)$ time using $\mathcal{O}(|\mathcal{D}|)$ memory.*

The above can be realised by associating each instance of the original dataset $\mathcal{D}$ with a unique ID and afterwards using direct hashing to query in constant time the presence of an

instance in a dataset. Once the differences have been computed, the frequency counts may be updated *incrementally*.

Proposition 8 *(Computational Complexity of Incremental Frequency Computation) Let m denote the maximum number of features in any considered instance. Given the frequency counts FQ_{old} of a previous dataset $\mathcal{D}_{old}$, a new dataset $\mathcal{D}_{new}$, and their differences $\mathcal{D}_{in}$ and $\mathcal{D}_{out}$, the frequency counts FQ_{new} of the new dataset $\mathcal{D}_{new}$ can be computed in $\mathcal{O}((|\mathcal{D}_{in}| + |\mathcal{D}_{out}|) \cdot m^2)$ time.*

To show the complexity, note the difference between FQ_{old} and FQ_{new}.

Observation 2 *Let $\mathcal{K}(FQ)$ denote the set of instances used to compute the frequency counts FQ. It follows that $\mathcal{K}(FQ_{old}) = \mathcal{D}_{out} \cup \mathcal{D}_{same}$ and $\mathcal{K}(FQ_{new}) = \mathcal{D}_{in} \cup \mathcal{D}_{same}$.*

Consider taking FQ_{old} and applying a series of operations to reach the new frequency counts FQ_{new}. The complexity result of Proposition 8 follows from the previous observations and the following:

Observation 3 *The frequency counts FQ_{old} already capture the counts for instances $\mathcal{D}_{same}$*

Observation 4 *The frequency counts FQ_{old} need to be incremented using instances $\mathcal{D}_{in}$*

Observation 5 *The frequency counts FQ_{old} need to be decremented using instances $\mathcal{D}_{out}$*

Using the incremental update procedure is sensible only if the number of updates required is small compared to recomputing from scratch. Therefore, in our framework, in each call to compute a decision tree of depth two, the algorithm incurs an overhead (Proposition 7) to compute the differences between the old and new dataset. It proceeds with the incremental computation if $|\mathcal{D}_{in} \cup \mathcal{D}_{out}| < |\mathcal{D}_{new}|$, and otherwise computes from scratch.

Note that the overhead is negligible compared to the overall complexity of computing the optimal tree of depth two (Proposition 6), but the benefits can be significant if the difference is small. As shown in the experimental section, this is frequently the case in practice, as two successive features considered for splitting are unlikely to lead to vastly different splits.

4.5 Similiarity-Based Lower Bounding

We present a novel lower bounding technique that does not rely on the algorithm having previously searched a specific path, as opposed to the cache-based lower bound introduced in the later sections. Given a dataset $\mathcal{D}_{new}$ for a node, the method aims to derive a lower bound by taking into account a previously computed optimal decision tree using the dataset $\mathcal{D}_{old}$. It infers the bound by considering the difference in the number of instances between the previous dataset $\mathcal{D}_{old}$ and the current dataset $\mathcal{D}_{new}$. The bound is used to prune portions of the search space that are guaranteed to not contain a better solution than the best decision tree encountered so far in the search.

Assume that for both datasets, the depth and the number of allowed feature nodes requirements are identical. As in the previous section, we define the sets $\mathcal{D}_{in} = \mathcal{D}_{new} \setminus \mathcal{D}_{old}$, $\mathcal{D}_{out} = \mathcal{D}_{old} \setminus \mathcal{D}_{new}$, and $\mathcal{D}_{same} = \mathcal{D}_{new} \cap \mathcal{D}_{old}$.

Given the limits on the depth d and number of features nodes n, a dataset $\mathcal{D}_{new}$, and a dataset $\mathcal{D}_{old}$ with $T(\mathcal{D}_{old}, d, n)$ as the misclassification score of the optimal decision tree of $\mathcal{D}_{old}$ (recall Eq. 1), we define the similarity-based lower bound,

$$LB(\mathcal{D}_{new}, \mathcal{D}_{old}, d, n) = T(\mathcal{D}_{old}, d, n) - |\mathcal{D}_{out}|, \tag{9}$$

which is a lower bound for the number of misclassifications of the optimal decision tree for the dataset $\mathcal{D}_{new}$ of a tree of depth d with n feature nodes, i.e.,

Proposition 9 $LB(\mathcal{D}_{new}, \mathcal{D}_{old}, d, n) \leq T(\mathcal{D}_{new}, d, n)$.

As a result, subtrees with a lower bound greater than its upper bound are immediately pruned, effectively speeding up the search. To show that Proposition 9 is indeed a lower bound, let $T(\mathcal{D}) = T(\mathcal{D}, d, n)$, note that removing $\mathcal{D}_{out}$ from $\mathcal{D}_{old}$ may reduce the misclassification cost by at most $|\mathcal{D}_{out}|$:

$$T(\mathcal{D}_{old}) - T(\mathcal{D}_{old} \setminus \mathcal{D}_{out}) = T(\mathcal{D}_{old}) - T(\mathcal{D}_{same}) \leq |\mathcal{D}_{out}|. \tag{10}$$

$$T(\mathcal{D}_{old}) - |\mathcal{D}_{out}| \leq T(\mathcal{D}_{same}). \tag{11}$$

Adding instances to $\mathcal{D}_{same}$ cannot decrease the misclassification score $T(\mathcal{D}_{same})$:

$$T(\mathcal{D}_{new}) = T(\mathcal{D}_{same} \cup \mathcal{D}_{in}) \geq T(\mathcal{D}_{same}) \tag{12}$$

Combining Eq. 11 and 12 we arrive at:

$$T(\mathcal{D}_{old}) - |\mathcal{D}_{out}| \leq T(\mathcal{D}_{new}) \tag{13}$$

$$LB(\mathcal{D}_{new}, \mathcal{D}_{old}, d, n) \leq T(\mathcal{D}_{new}) \tag{14}$$

which shows the derivation of Proposition 9.

As shown in the experimental results (Section 5.2.1), the use of the similarity-based lower bound reduces the runtime for all datasets, with only a few exceptions, and in some cases the obtained reduction is an order of magnitude.

4.6 Caching of Optimal Subtrees (Memoisation)

As is common in dynamic programming algorithms, a caching or memoisation table is maintained to avoid recomputing subproblems. In our case, information about computed optimal subtrees is stored. This is used to retrieve a subtree that has already been computed when needed, provide lower bounds, and reconstruct the optimal decision tree at the end of the algorithm. Caching has been used in previous works (Nijssen and Fromont (2007); Aglin et al. (2020); Verhaeghe et al. (2019); Hu et al. (2019)), and here we extend it to support constraints on the number of nodes and incremental solving.

The key observation is that given a path from the root to any given node, each permutation of the feature nodes on the path results in the same dataset for the node furthest from the root, e.g., $\mathcal{D}(f_i)(\overline{f_j}) = \mathcal{D}(\overline{f_j})(f_i)$. This allows representing a path as a set of features,

e.g., $\{f_i, \overline{f_j}\}$. Each time an optimal subtree is computed during search, its path and root node are stored in the cache. If no subtree could be computed within the specified upper bound, a lower bound is derived based on the information collected during the search and the given upper bound. The bound is stored in the cache for reuse later on. Caching and lower bounds derived in this manner have been used in DL8.5 Aglin et al. (2020) to compute optimal decision trees with a given depth.

We generalise caching to support being used when the number of nodes is also a constraint, in addition to the depth, and allow incremental solving, i.e., the scenario when progressively large trees in terms of depth and size are computed in succession, e.g., during hyper-parameter tuning. Furthermore, we strengthen the bound introduced in DL8.5 (Aglin et al. (2020)) by using information obtained during the search.

Recall that only the root node is stored for a given path. When necessary, the complete subtree may be reconstructed as a series of queries to the cache, where each time a single node is retrieved, as introduced in DL8 (Nijssen and Fromont (2007)). In our framework, there is an exception to the mentioned tree reconstruction procedure. After solving a tree of depth two, none of its children are stored in the cache. During the algorithm these are not necessary, but the children are needed when reconstructing the best decision tree found at the end. In this case, the required child nodes are recomputed using Algorithm 3. The computational overhead is negligible compared to the overall execution time, but this avoids storing an exponential number of paths (recall that the number of paths increases exponentially with the depth) which do not serve a purpose other than the final reconstruction.

4.6.1 Storing Subtrees and Lower Bounds in the Cache

Each node is associated with a path, represented as a set of features. Our cache maps a path to a list of cache entries, where each entry is composed of an optimal assignment, a lower bound, and the size limit. The depth is not explicitly stored as it can be derived based on the maximum depth and the number of features in the path. It is possible to store a lower bound without the optimal assignment, but note that the optimal assignment is the tightest lower bound. Initially, the cache is empty, and the lists and their entries are created dynamically during search as needed. There are two types of scenarios that prompt a cache storage.

Scenario one: node has been exhaustively searched and a solution has been found within the upper bound. In this situation, the computed subtree is optimal and the corresponding entry is stored using its root node assignment as the optimal assignment, the lower bound is set to the misclassification score, and the feature node limit is the limit that was assigned to the node.

In case the algorithm determines that the lowest classification score may be achieved using fewer nodes than imposed by the node limit, we may use the following proposition to create additional cache entries:

Proposition 10 *Let $T(\mathcal{D}, d, n)$ be the misclassification score of the optimal decision tree for the dataset $\mathcal{D}$ with depth limit d and node limit n. If there exists an $n' < n$ such that $T(\mathcal{D}, d, n') = T(\mathcal{D}, d, n)$, then $T(\mathcal{D}, d, i) = T(\mathcal{D}, d, n)$ for $i \in [n', n]$.*

During the algorithm, a given path can only be exhaustively explored once, as the next time the path is encountered its corresponding solution is retrieved from the cache (Section 4.6.2).

Scenario two: a node has been exhaustively explored, but no solution has been found within the upper bound. It follows that the optimal subtree corresponding to the path has at least as many misclassifications as the upper bound incremented by one. This is the lower bounding reasoning introduced in DL8.5 (Aglin et al. (2020)).

In this work, we propose a stronger lower bound. Let $LB(\mathcal{D}, d, n)$ be the lower bound for the number of misclassifications of an optimal decision tree for the dataset $\mathcal{D}$ with n nodes and depth d, i.e., $T(\mathcal{D}, d, n) \geq LB(\mathcal{D}, d, n)$. We introduce the following refined lower bound RLB:

$$RLB(\mathcal{D}, d, n) = \min\{LB(\mathcal{D}(\overline{f}, d-1, s_1) + LB(\mathcal{D}(f, d-1, s_2) \mid f \in \mathcal{F} \wedge s_1 + s_2 = n-1\} \quad (15)$$

The bound RLB considers all possible assignments of features and sizes to the root and its children, and selects the minimum sum of the lower bounds of its children. It follows that no decision tree may have a misclassification score lower than RLB. We combine RLB with the upper bound to obtain a lower bound for the case where no decision tree with less than the specified upper bound UB could be found:

$$T(\mathcal{D}, d, n) \geq max\{RLB(\mathcal{D}, d, n),\ UB + 1\}. \quad (16)$$

The proposed bound generalises the bound from DL8.5 (Aglin et al. (2020)), which only considers the second expression on the right-hand side of Eq. 16 to derive a lower bound when no tree could be found within the given upper bound. In our experiments, we observed the strengthened bound provides a speed-up by a factor of at most two on several datasets.

Once the bound has been computed, it is recorded in a cache entry for the path, along with the size limit of the node, and the optimal assignment is set to null.

4.6.2 Retrieving Subtrees and Lower Bounds from the Cache

When a new child node is created, the algorithm searches through the cache to detect if the optimal solution is already present. Ideally, the cache entry of the path matches the size limit imposed on the node, but a lower bound for the current tree may be inferred from the bounds of the larger tree, formally summarised in the following proposition.

Proposition 11 *Given the dataset $\mathcal{D}$ and depth bound d and the maximum number of feature nodes n, a bound for a larger tree is a bound for the current tree, i.e., $\forall n' \geq n, d' \geq d : LB(\mathcal{D}, d', n') \leq LB(\mathcal{D}, d, n)$.*

When retrieving a lower bound and no lower bound has been stored for the currently queried decision tree, Proposition 11 allows inferring a lower bound from larger trees that may be stored in the cache. Note that the lower bounds are nonincreasing with size, i.e.,

$$LB(\mathcal{D}, d, n) \leq LB(\mathcal{D}, d, n+1) \quad (17)$$

The tightest bound is returned when retrieving the lower bound. Assuming no bound with the prescribed number of nodes is stored, the bound of the smallest tree that is greater than the target size is returned should such an entry exist. If there are no applicable entries in the cache, the trivial lower bound of zero is returned. For example, given a dataset $\mathcal{D}(f_1, f_2)$ with the node limit set to five, if there is no subtree for the given size in the cache but there is an entry when the node limit was set to six and seven, then the lower bound using six nodes is the tightest valid lower bound available for the tree with five nodes.

4.6.3 Incremental Solving

We label *incremental solving* as the process of querying the algorithm to compute progressively larger decision trees. For example, once the algorithm has computed an optimal decision tree for a given depth and size, a user may be interested in a tree with more nodes to understand if the additional nodes would lead to a meaningful decrease in the misclassification score, or as part of hyper-parameter tuning.

The cache naturally supports these types of queries when the depth is fixed and the size is increased, since the problem of computing a larger tree includes smaller trees as its subproblems, and all cache entries remain valid. In a sense, the framework incorporates incremental solving throughout its execution. However, not all entries can be kept once the global depth is increased.

When search and caching is performed, the results learned are (*implicitly*) only correct with respect to the maximum depth as the depth is not *explicitly* stored. Nevertheless, a portion of the cache entries remains valid when the depth is increased. Observe that for certain size values, increasing the depth is redundant and does not increase the search space. Intuitively, the limited tree size does not allow benefiting from a larger depth. Formally, given size s and depth d, if $s \leq d$, then the set of all possible decision trees of size s with depth d is equivalent to the set of decision trees of size s with depth d', where $d' \geq d$. As a result, when incrementally computing a globally optimal decision tree of depth d' and the cache of the computation of a tree of depth d is available with $d' > d$, we keep all cache entries such that $|P| + size\_limit \leq d$, where P is the corresponding path of the entry, and discard the rest.

4.7 Node Selection Strategy

Given a feature for a node and the size allocation for its children, the algorithm decides on which child node to recurse on first. Our search strategy is a variant of post-order traversal, labelled *dynamic post-order*, which dynamically decides which child node to visit first. The idea is to prioritise the child node that has the heuristically-determined largest potential to improve the current decision tree, which in turn leads to a higher chance to prune to the other sibling. The potential is computed as the gap between the upper and lower bound of the child. Note that the upper bound of the parent is used as the upper bound of its children. The lower bound for a child is retrieved from the cache. This provided consistent improvements in runtime (Section 5.2.3) when compared to the strategy used in DL8.5 (Aglin et al. (2020)), which always visits the left subtree before the right subtree.

4.8 Feature and Size Selection

For a given node, each possible tree configuration (a feature and the size of its children) is considered one at a time, unless the node is pruned or the optimal solution is retrieved from the cache (see Subsection 4.6). The order in which tree configurations are explored may have an impact on performance, as evidenced in most search algorithms in general.

In our experiments, the simplest variants performed the best: features are selected in the order as given in the dataset, and the size is distributed by considering increasingly larger right subtrees as described in Algorithm 1. Alternative options are possible, such as ordering the features according to their corresponding Gini coefficient as done in heuristic algorithms, but none of these alternatives lead to sizable benefits over the simplest strategies. This is discussed in more detail in the experimental section (Section 5.2.3).

4.9 Extensions

To ease the presentation, we discussed our framework in the context of binary classification on binary datasets. In this section, we discuss extensions to general settings.

4.9.1 General Data

The input to our framework is a dataset containing binary features. Datasets with continuous and/or categorical features are binarised. In our work, this done using a supervised discretisation algorithm based on the Minimum Description Length Principle (MDLP) by Fayyad and Irani (1993), effectively converting each feature into a categorical feature based on the statistical significance of the feature values for the class, and then using a one-hot encoding to binarise the features. We use the MDLP implementation available in the R programming package (Kim (2015)).

Note that (univariate) decision tree algorithms *implicitly* binarise the dataset during learning. Each feature node is assigned a predicate that evaluates whether a given feature value meets a threshold, which can be seen as a binary feature. When binarising the dataset, the possible decisions are decided upfront rather than during execution.

4.9.2 Multi-Classification

To extend the algorithm for multi-classification, the key step is to generalise Algorithm 3 to compute frequency counters for each class. Equations analogous to Equations 2—8 are devised to compute the misclassification scores. Since classes partition the data, the complexity results remain valid for multi-classification.

4.9.3 Additional Constraints and Objective Functions

Our framework may be modified to support constraints and objective functions that can be expressed in the dynamic programming formulation (Eq. 1) and solved using a (variant) of the specialised algorithm (Section 4.3).

Furthermore, objective functions that have a dependency on the number of nodes can also be handled. An alternate objective for optimal decision trees is

$$misclassifications + \alpha \times nodes \tag{18}$$

which balances the size of the decision tree against the misclassifications. This objective was used in the original CART paper (Breiman et al. (1984)) and discussed in some of the other optimal decision tree works (Bertsimas and Dunn (2017); Hu et al. (2019)). While we could extend our solution to directly work with the sparse objective, we implemented a non-intrusive modification. We compute a series of decision trees minimising the misclassification score using k nodes, where $k \in [0, max\_num\_nodes]$, and then evaluate each solution with respect to the objective and select the best. Our framework supports incremental solving, allowing cached subtrees from computing a decision tree with k' nodes to be reused when considering $k' + 1$ nodes. Another way of viewing the process is to consider it as a hyper-parameter tuning procedure where the resulting trees are evaluated with respect to the new objective. A similar procedure was used by Bertsimas and Dunn (2017) to tune their integer programming approach.

We discussed multi-classification and the above objective function as special cases. Additional examples may include imposing a minimum number of instances per node, different linear penalties for misclassifying classes rather than treating each misclassification equally, and adding fairness objectives and constraints as in the work of Aghaei et al. (2019).

5. Computational Study

The goal of this section is to compare the performance of our method with the state-of-the-art and empirically evaluate the scalability of our algorithm and the benefits of incremental computation and lower bounding. With this in mind, we designed three major themes to investigate, each addressing a unique set of questions: variations and scalability of our approach, effectiveness compared to the state-of-the-art optimal classification tree algorithms, and out-of-sample accuracy as compared to heuristically-obtained decision trees and random forests.

5.1 Datasets and Computational Environment

We use publicly available datasets used in previous works (Bertsimas and Dunn (2017); Verwer and Zhang (2019); Narodytska et al. (2018); Aglin et al. (2020); Hu et al. (2019)), most of which are available from the UCI and CP4IM repositories. The datasets include 85 classification problems with a mixture of binary, categorical, and continuous features. Datasets with categorical and/or continuous features are converted into binary datasets as discussed in Section 4.9.1. Datasets with missing values were excluded from experimentation. We note that some benchmarks appeared in previous works under different binarisation techniques or simplifications, e.g., multi-classification turned into binary-classification using a 'one-versus-all' scheme or a subset of the features were removed. For these cases, we include the different versions as separate datasets. The binarised datasets together with the binarisation script will be readily available soon (please contact the first author in the meantime).

Experiments were run on an Intel i-7-7700HQ CPU with 32 GB of RAM running one algorithm at a time using one processor. The timeout was set to ten minutes except for the hyper-parameter tuning where no limit was enforced. In the following, we dedicate a separate subsection to each of the three major experimental topics.

5.2 Variations of Our Algorithm and Scalability

The aim of this subsection is to investigate variations of our approach, determine the effectiveness of the introduced incremental computation and lower bounding techniques, analyse scalability, and impact of the feature and node selection strategies.

In the first part, we consider the algorithm without incremental or similarity-based lower bound computation, and observe the effect of adding these techniques to the runtime. In the second part, we discuss the impact of the number of instances, depth, and features with respect to the runtime. In the third part, we fix the algorithm parameters to their default settings and vary either the feature or node selection strategy.

We note that the default setting of our algorithm uses all techniques presented in the paper and the in-order feature selection and dynamic post-order node selection strategy.

5.2.1 Part One: Incremental and Lower Bound Computation

In Table 1, we show the effect of incrementally computing the frequency counters (Section 4.4.3) rather than doing it from scratch in each iteration, and combining it with our similarity-based lower-bound (Section 4.5).

The trend across all benchmarks is uniform, as each addition to the algorithm improves the runtime considerably. Incremental computation is useful as the splits considering two features might only differ in a small number of instances. Thus performing minor changes to the previously computed frequency counters is more favourable than recomputing from scratch. For similar reasons, the lower bound works well as it manages to identify cases where computing a subtree is unnecessary unless it deviates enough from the previously computed subtree.

An important observation that contributes to these positive points is that computing the difference of two sets of instances can be done in linear time with respect to the number of instances, which is comparatively inexpensive when considering the time spent on computing optimal subtrees of depth two using the specialised algorithm, but can save a significant amount of computation.

5.2.2 Part Two: Scalability

We investigate the sensitivity of our algorithm with respect to the number of instances and maximum depth. In Table 2, results are shown when our algorithm is run to compute trees of $\mathit{depth} \in [4, 5]$ on datasets where instances are duplicated $k \in [1, 2, 4]$ times. We note that trees with depth three are omitted as these are computed within seconds. The results indicate a linear dependency with the number of instances for the majority of the datasets. As most of the computational time is spent in repeatedly solving optimal subtrees of depth two (Section 4.4), the finding is consistent with the theoretical complexity (Proposition 6). This is a notable improvement over generic optimisation approaches, such as integer programming or SAT. The latter may exhibit an exponential runtime dependency on the number of instances as new binary variables are introduced for each instance, and typically do not consider datasets with more than a thousand instances.

Note that the experiments regarding the scalability with respect to the number of instances are merely indicative. In practice, however, introducing more instances might implicitly increase or decrease the number of binary features in the discretisation and have

Table 1: Comparison to determine effectiveness of the incremental frequency counter computation and the similarity-based lower bound. The baseline excludes both techniques. For each dataset, the number of instances ($\mathcal{D}$), number of binary features ($\mathcal{F}$), and number of classes ($\mathcal{C}$) are displayed. Entries display runtime in seconds (rounded to nearest integer) to compute the optimal classification tree of depth four. Time limit set to ten minutes. Timeouts denoted as —. Datasets solved under half a second by all variants have been omitted.

Name	$\lvert\mathcal{D}\rvert$	$\lvert\mathcal{F}\rvert$	$\lvert\mathcal{C}\rvert$	OurBaseline	IncFQ	IncFQ+SimilarLB
anneal	812	93	2	4	1	**< 0.5**
audiology	216	148	2	7	1	**1**
australian-credit	653	125	2	10	3	**2**
breast-wisconsin	683	120	2	8	2	**1**
diabetes	768	112	2	11	4	**3**
german-credit	1000	112	2	7	3	**3**
heart-cleveland	296	95	2	2	**1**	**1**
hypothyroid	3247	88	2	12	3	**2**
ionosphere	351	445	2	—	201	**101**
kr-vs-kp	3196	73	2	7	2	**1**
letter	20000	224	2	—	515	**333**
mushroom	8124	119	2	2	**1**	**1**
pendigits	7494	216	2	—	172	**95**
splice-1	3190	287	2	300	168	**163**
vehicle	846	252	2	180	36	**12**
yeast	1484	89	2	8	3	**2**
biodeg	1055	81	2	2	**1**	**1**
default_credit	30000	44	4	6	**5**	7
HTRU_2	17898	57	2	**2**	**2**	3
Ionosphere	351	143	2	4	**2**	**2**
magic04	19020	79	2	6	**4**	6
spambase	4601	132	2	27	11	**11**
Statlog_satellite	4435	385	6	—	611	**276**
Statlog_shuttle	43500	181	7	130	72	**63**
appendicitis-un	106	530	2	239	212	**9**
australian-un	690	1163	2	—	—	**530**
backache-un	180	475	2	163	143	**12**
cancer-un	683	89	2	1	**< 0.5**	**< 0.5**
cleve-un	303	395	2	79	70	**10**
colic-un	368	415	2	111	91	**22**
heart-statlog-un	270	381	2	68	59	**8**
hepatitis-un	155	361	2	55	47	**6**
hungarian-un	294	330	2	39	34	**6**
new-throid-un	215	334	3	125	117	**13**
promoters-un	106	334	2	26	21	**2**
shuttleM-un	14500	691	2	—	—	**169**

an effect on shaping the structure of the dataset, both of which may impact positively or negatively the running time. The results do show that the bottleneck of the approach is not necessarily in the number of instances.

In contrast to the number of instances, the depth consistently has a large impact on the running time. The number of possible decision trees grows exponentially as the depth increases, which is reflected in the computational experiments. For example, our approach computes depth-three trees within seconds, but the runtimes go up notably for depth four and five. In previous works, depth four has been seen as the benchmark value, but with

MurTree such trees are computable within a reasonable time, and in many cases greater depths are also possible.

Apart from the depth, another important factor is the number of binary features, which additionally dictates the number of possible decision trees necessary to explore to find the optimal tree. As the ability of our techniques to prune and reduce computational time depends on the structure of the dataset, it is difficult to artificially increase the number of features and show the dependency. For example, duplicating features would not lead to conclusive statements on the impact of the number of features on runtime, as our lower bounding mechanism would trivially prune these features. We instead refer to the computational complexity of our algorithm from Proposition 6 and the number of possible decision trees as an indicative measure of the influence of the number of binary features and sparsity of the feature vectors on the runtime. Note that some of the instances contain a high number of features, e.g., australian-un has 1163 binary features, but it is still within practical reach.

Overall, we conclude that our approach scales reasonably well for the tested datasets with depth four, having computed the optimal decision tree for the majority of the instances within seconds, and all within ten minutes. Deeper optimal decision trees of depth five or greater remain a challenge.

5.2.3 Part Three: Feature and Node Selection Strategies

In Tables 3, we vary the feature and node selection strategies as presented in Section 4.8 for decision trees of depth four with fifteen decision nodes.

The best performing feature selection strategy is the in-order variant, which selects features in the order given in the dataset. The benefit is that there is no additional overhead introduced and it orders the features in a manner that the incremental computation and similarity-based lower bounding can exploit, as two neighbouring features in an instance tend to be similar, which is a result of binarisation. In contrast, using the Gini measure for feature ordering can be beneficial in finding a good decision tree early in the search, but it carries an overhead for each node which does not pay off in proving optimality.

The dynamic post-order strategy provides consistent improvement over the fixed post-order strategy used by DL8.5 (Aglin et al. (2020)). Given a node, once one of its children have been exhaustively explored, tighter upper bounds can be imposed on the other sibling. As this is the main mechanism for pruning, it is important to direct the algorithm to exhaust a child node as soon as possible. Heuristically speaking, solving the subtree with a greater potential for misclassification can yield tighter constraints on the other sibling once it has been exhaustively explored, resulting in more frequent pruning.

To summarise this section, the experimental results have confirmed the efficiency of our incremental frequency computation and similarity-based lower bounding approach. Each of the techniques provides a reduction in terms of runtime. Our approach scales (roughly) linearly with respect to the number of instances, and the depth of the tree has a large influence on the runtime, i.e., decision trees of depth three and four are typically computed within seconds or minutes, respectively, but trees of depth five are notably more challenging as they may timeout after ten minutes, depending on the dataset. Increasing the number of binary features increases the expected runtime, but this is difficult to measure it depends

Table 2: Scalability of MurTree as instances are duplicated two and four times. For each dataset, the number of instances ($\mathcal{D}$), number of binary features ($\mathcal{F}$), and number of classes ($\mathcal{C}$) are displayed. Entries display runtime in seconds (rounded to nearest integer) to compute the optimal classification tree of depth four and five. Time limit set to ten minutes. Timeouts denoted as — Datasets solved under a second and a half by all variants have been omitted.

Name	$\lvert\mathcal{D}\rvert$	$\lvert\mathcal{F}\rvert$	$\lvert\mathcal{C}\rvert$	Depth=4			Depth=5		
				$\mathcal{D}$	$2\cdot\mathcal{D}$	$4\cdot\mathcal{D}$	$\mathcal{D}$	$2\cdot\mathcal{D}$	$4\cdot\mathcal{D}$
anneal	812	93	2	< 0.5	1	1	5	9	16
australian-credit	653	125	2	2	3	6	63	92	162
breast-wisconsin	683	120	2	1	1	2	2	4	6
diabetes	768	112	2	3	4	8	101	160	296
german-credit	1000	112	2	3	5	9	111	187	331
heart-cleveland	296	95	2	1	1	1	9	13	20
hypothyroid	3247	88	2	2	4	9	48	95	186
ionosphere	351	445	2	101	163	279	215	336	623
kr-vs-kp	3196	73	2	1	3	5	24	46	92
letter	20000	224	2	333	—	—	—	—	—
mushroom	8124	119	2	1	2	4	1	2	4
pendigits	7494	216	2	95	166	314	415	—	—
soybean	630	50	2	< 0.5	< 0.5	< 0.5	1	1	2
splice-1	3190	287	2	163	297	572	—	—	—
tic-tac-toe	958	27	2	< 0.5	< 0.5	< 0.5	1	1	2
vehicle	846	252	2	12	21	44	446	—	—
vote	435	48	2	< 0.5	< 0.5	< 0.5	1	1	2
yeast	1484	89	2	2	4	6	77	136	226
fico-binary	10459	17	2	< 0.5	< 0.5	< 0.5	2	4	8
bank_conv	4521	26	2	< 0.5	1	1	2	3	7
biodeg	1055	81	2	1	2	4	32	53	90
default_credit	30000	44	4	7	19	98	114	322	—
HTRU_2	17898	57	2	3	7	24	65	141	486
Ionosphere	351	143	2	2	2	3	3	4	7
magic04	19020	79	2	6	21	69	202	474	—
spambase	4601	132	2	11	23	48	351	—	—
Statlog_satellite	4435	385	6	276	353	527	—	—	—
Statlog_shuttle	43500	181	7	63	320	—	—	—	—
appendicitis-un	106	530	2	9	10	11	—	—	—
australian-un	690	1163	2	530	563	—	—	—	—
backache-un	180	475	2	12	12	12	231	241	260
cancer-un	683	89	2	< 0.5	1	1	9	13	22
cleve-un	303	395	2	10	11	13	—	—	—
colic-un	368	415	2	22	25	29	—	—	—
haberman-un	306	92	2	< 0.5	< 0.5	1	7	9	13
heart-statlog-un	270	381	2	8	8	10	—	—	—
hepatitis-un	155	361	2	6	7	7	334	366	436
hungarian-un	294	330	2	6	7	8	431	499	—
new-throid-un	215	334	3	13	14	15	—	—	—
promoters-un	106	334	2	2	2	3	2	2	2
shuttleM-un	14500	691	2	169	358	—	—	—	—

on the effectiveness of the pruning techniques for the dataset at hand. Lastly, we found that inspecting features in the order as given in the dataset was more effective than ordering features according to their corresponding Gini coefficients, and our dynamic node selection strategy offered consistent improvements over a static strategy.

Table 3: Comparison of our default approach (dynamic post-order node selection and in-order feature selection) with a fixed post-order strategy and a feature selection strategy in decreasing Gini coefficients. For each dataset, the number of instances ($\mathcal{D}$), number of binary features ($\mathcal{F}$), and number of classes ($\mathcal{C}$) are displayed. Entries display runtime in seconds (rounded to nearest integer) to compute the optimal classification tree of depth four. Datasets solved under a second or when the differences between the variants was negligible have been omitted.

Name	$\lvert\mathcal{D}\rvert$	$\lvert\mathcal{F}\rvert$	$\lvert\mathcal{C}\rvert$	PostOrder	Gini	Default
anneal	812	93	2	1	1	**< 0.5**
audiology	216	148	2	**1**	**1**	**1**
australian-credit	653	125	2	**2**	4	**2**
breast-wisconsin	683	120	2	**1**	2	**1**
diabetes	768	112	2	**3**	4	**3**
german-credit	1000	112	2	**3**	4	**3**
heart-cleveland	296	95	2	**1**	**1**	**1**
hypothyroid	3247	88	2	**2**	3	**2**
ionosphere	351	445	2	149	251	**101**
kr-vs-kp	3196	73	2	2	2	**1**
letter	20000	224	2	459	603	**333**
mushroom	8124	119	2	**1**	**< 0.5**	**1**
pendigits	7494	216	2	96	177	**95**
splice-1	3190	287	2	167	112	**163**
vehicle	846	252	2	19	30	**12**
yeast	1484	89	2	**2**	3	**2**
biodeg	1055	81	2	**1**	2	**1**
default_credit	30000	44	4	9	9	**7**
HTRU_2	17898	57	2	**3**	4	**3**
Ionosphere	351	143	2	**2**	**2**	**2**
magic04	19020	79	2	7	10	**6**
spambase	4601	132	2	17	13	**11**
Statlog_satellite	4435	385	6	294	**209**	276
Statlog_shuttle	43500	181	7	**60**	124	63
appendicitis-un	106	530	2	**9**	35	**9**
australian-un	690	1163	2	556	639	**530**
backache-un	180	475	2	**11**	18	12
cancer-un	683	89	2	**< 0.5**	1	**< 0.5**
cleve-un	303	395	2	11	15	**10**
colic-un	368	415	2	23	33	**22**
heart-statlog-un	270	381	2	**8**	15	**8**
hepatitis-un	155	361	2	7	12	**6**
hungarian-un	294	330	2	7	11	**6**
new-throid-un	215	334	3	14	16	**13**
promoters-un	106	334	2	2	**< 0.5**	2
shuttleM-un	14500	691	2	197	386	**169**

5.3 Comparison Against State-Of-The-Art Optimal Decision Tree Algorithms

Amongst the optimal decision tree methods discussed in Section 3, we consider DL8.5 by Aglin et al. (2020) as the main competing method. The rationale is that DL8.5 has been shown to largely outperform the other techniques based on generic optimisation modelling, such as integer programming (Verwer and Zhang (2019); Bertsimas and Dunn (2017)) and constraint programming (Verhaeghe et al. (2019)), when minimising the misclassification score given constraints on the depth of the tree. There are two other approaches worth mentioning as a direct comparison was not considered before in the literature.

The SAT method by Narodytska et al. (2018) takes a different approach: rather than directly minimising the misclassifications given a fixed depth, it attempts to construct the smallest decision in terms of the total number of nodes that *perfectly* fits the data, i.e., trees that have a misclassification score of zero. As finding the zero-misclassification tree using the complete dataset was computationally infeasible for SAT, and also prone to overfitting, the authors proposed to subsample datasets by selecting 5-20% of the instances. While this setting has its merits, it diverges from the goals of our paper. Furthermore, we found that our algorithm computes the perfect decision tree within seconds on exactly the same subsampled data used in the SAT paper and as can be seen in tables, we can directly optimise with the complete datasets. The reason for the discrepancy in runtime between our approach and SAT is that we provide a highly specialised procedure that exploits classification tree properties, e.g., Properties 1 and 2. The same reasoning holds when comparing to other generic (optimisation) frameworks such as integer programming. For these reasons, we decided not to further discuss the SAT method.

Hu et al. (2019) introduced an algorithm for minimising the weighted sum of the number of misclassifications and number of nodes (Eq. 18). We may support this objective as part of hyper-parameter tuning (see Section 4.9.3). We found that our framework computes optimal trees with the specified objective within seconds for the benchmarks used by Hu et al. (2019). We are also able to optimise with solely the misclassification criteria, whereas their algorithm relies on the sparsity weight α having a significant impact on the optimality criteria as the main pruning technique is a bound based on α, e.g., unless α is sufficiently high, the approach does not terminate within a reasonable time. Lastly, their method was designed for up to twelve binary features, while in our experiments, the datasets may have up to one thousand binary features. Therefore, we do not do further comparisons as their approach times out on the majority of the datasets. The runtimes given in the hyper-parameter tuning section provide an insight into the computation time taken by our method to compute the optimal tree with the sparse objective (Eq. 18). We note that better runtimes could be obtained through modifying our algorithm to directly support the objective since additional pruning can be done.

5.3.1 Comparison with DL8.5 by Aglin et al. (2020)

The aim of this section is to assess the effectiveness of our MurTree approach with respect to DL8.5, the state-of-the-art method for optimal decision trees. In machine learning, it is standard practice to compare learning algorithms on out-of-sample accuracy. In our case, we evaluate the algorithms solely based on runtime. This is motivated by the fact that both algorithms are directly optimising the number of misclassifications. Therefore,

it makes sense to use runtime as the metric to assess the ability of the approaches in exhaustively exploring the search space to compute the provably optimal classification tree. A lower runtime indicates a more effective approach. We note that there may be several optimal decision trees with the same minimum misclassification score, but we do not discuss these in more detail as neither approach discriminates between them, i.e., trees are only evaluated based on their misclassification score. Analysing the particular differences among optimal decision trees is out of the scope of this work. Such an evaluation is tied to a broader question of designing a more appropriate objective function for decision trees to allow better generalisation. As out-of-sample accuracy is an important question for any machine learning algorithm, we reserved the next section to evaluate the quality of optimal decision trees in their current form as out-of-sample classifiers. We now proceed with the comparison with DL8.5.

We set the maximum depth to four for both methods. Since DL8.5 does not support constraining the number of nodes, the maximum number of feature nodes is set to fifteen for our method to ensure a fair comparison. We provide the complete dataset to both algorithms without dividing into the training and test set. Ten minutes is allocated for each dataset.

The runtimes, given in Table 4, show that our method is orders of magnitude faster than DL8.5. This is a significant result, as DL8.5 has been previously shown to outperform other techniques for optimal classification trees based on integer and constraint programming by a large margin. This further illustrates the advantage of designing and specialising domain-specific optimisation algorithms compared to using off-the-shelf tools. Both DL8.5 and our MurTree approach exploit the dynamic programming structure of decision trees, but our method employs additional techniques to further take advantage of the properties of decision trees. The reduced runtime contributes greatly towards the application of optimal classification tree methods in practice, especially when tuning is involved (see further for different tuning settings).

5.4 Comparison Against Conventional Algorithms on Out-Of-Sample Accuracy

In this section, we analyse the suitability of our optimal decision trees as out-of-sample classifiers. The aim is to demonstrate that more accurate trees of limited size lead to better generalisations than what is offered by heuristic approaches. Note that the restricted size of optimal decision trees plays the role of a regulariser to avoid overfitting. The main comparison is done against an optimised implementation of CART (Breiman et al. (1984)), a widely used decision tree learning algorithm. For illustrative purposes, we also make a comparison with random forests, as a related method that typically improves accuracy over standard decision tree algorithms at the expense of being less interpretable. As will be discussed, our experiments further confirm similar empirical findings (Verwer and Zhang (2019); Bertsimas and Dunn (2017)).

We perform grid-search hyper-parameter tuning for each method using stratified five-fold cross-validation, and each method is tuned and evaluated on exactly the same folds. We use the framework provided by the *sklearn* (Pedregosa et al. (2011)) Python package for

Table 4: Comparison of DL8.5 (Aglin et al. (2020)) and our approach, MurTree. For each dataset, the number of instances ($\mathcal{D}$), number of binary features ($\mathcal{F}$), and number of classes ($\mathcal{C}$) are displayed. Entries display runtime in seconds (rounded to nearest integer) to compute the optimal classification tree of depth four and five. Time limit set to ten minutes. Timeouts denoted as —

Name	$\mathcal{D}$	$\mathcal{F}$	$\mathcal{C}$	Depth=4		Depth=5	
				DL8.5	MurTree	DL8.5	MurTree
anneal	812	93	2	103	**< 0.5**	—	**5**
audiology	216	148	2	151	**1**	1	**< 0.5**
australian-credit	653	125	2	—	**2**	—	**63**
breast-wisconsin	683	120	2	396	**1**	—	**2**
diabetes	768	112	2	—	**3**	—	**101**
german-credit	1000	112	2	641	**3**	—	**111**
heart-cleveland	296	95	2	248	**1**	—	**9**
hepatitis	137	68	2	27	**< 0.5**	66	**< 0.5**
hypothyroid	3247	88	2	287	**2**	—	**48**
ionosphere	351	445	2	—	**100**	—	**215**
kr-vs-kp	3196	73	2	138	**2**	—	**24**
letter	20000	224	2	—	**333**	—	—
lymph	148	68	2	16	**< 0.5**	14	**< 0.5**
mushroom	8124	119	2	91	**1**	74	**1**
pendigits	7494	216	2	—	**95**	—	**415**
primary-tumor	336	31	2	4	**< 0.5**	23	**< 0.5**
segment	2310	235	2	5	**< 0.5**	3	**< 0.5**
soybean	630	50	2	7	**< 0.5**	77	**1**
splice-1	3190	287	2	—	**163**	—	—
tic-tac-toe	958	27	2	3	**< 0.5**	16	**1**
vehicle	846	252	2	—	**12**	—	**446**
vote	435	48	2	10	**< 0.5**	55	**1**
yeast	1484	89	2	434	**2**	—	**77**
compas-binary	6907	12	2	2	**< 0.5**	2	**< 0.5**
fico-binary	10459	17	2	3	**< 0.5**	15	**2**
balance-scale	625	4	3	1	**< 0.5**	1	**< 0.5**
banknote	1372	16	2	5	**< 0.5**	38	**< 0.5**
bank_conv	4521	26	2	1	**< 0.5**	2	**2**
biodeg	1055	81	2	130	**1**	—	**32**
car_evaluation	1728	7	4	1	**< 0.5**	1	**< 0.5**
default_credit	30000	44	4	484	**7**	—	**114**
HTRU_2	17898	57	2	190	**3**	—	**65**
IndiansDiabetes	768	11	2	1	**< 0.5**	1	**< 0.5**
Ionosphere	351	143	2	230	**2**	549	**3**
iris	150	12	3	1	**< 0.5**	1	**< 0.5**
magic04	19020	79	2	—	**6**	—	**202**
messidor	1151	24	2	2	**< 0.5**	11	**< 0.5**
monk1_bin	124	15	2	1	**< 0.5**	1	**< 0.5**
monk2_bin	169	15	2	2	**< 0.5**	1	**< 0.5**
monk3_bin	122	15	2	1	**< 0.5**	1	**< 0.5**
seismic_bumps	2584	10	2	2	**< 0.5**	1	**< 0.5**
spambase	4601	132	2	—	**11**	—	**351**
Statlog_satellite	4435	385	6	—	**276**	—	—
Statlog_shuttle	43500	181	7	—	**64**	—	—
tic-tac-toe_bin	958	18	2	2	**< 0.5**	4	**< 0.5**
winequality-red	1599	17	6	1	**< 0.5**	1	**< 0.5**
wine	178	32	3	2	**< 0.5**	5	**< 0.5**
appendicitis-un	106	530	2	—	**9**	—	—
australian-un	690	1163	2	—	**530**	—	—
backache-un	180	475	2	—	**12**	—	**231**
cancer-un	683	89	2	43	**< 0.5**	—	**9**
car-un	1728	21	2	2	**< 0.5**	5	**< 0.5**
cleve-un	303	395	2	—	**10**	—	—
colic-un	368	415	2	—	**23**	—	—
corral-un	160	6	2	1	**< 0.5**	1	**< 0.5**
haberman-un	306	92	2	34	**< 0.5**	—	**7**
heart-statlog-un	270	381	2	—	**8**	—	—
hepatitis-un	155	361	2	—	**6**	—	**334**
house-votes-84-un	435	16	2	2	**< 0.5**	2	**< 0.5**
hungarian-un	294	330	2	—	**6**	—	**431**
irish-un	500	112	2	4	**< 0.5**	4	**< 0.5**
mouse-un	70	45	2	1	**< 0.5**	1	**< 0.5**
mux6-un	128	6	2	1	**< 0.5**	1	**< 0.5**
new-throid-un	215	334	3	—	**13**	—	—
promoters-un	106	334	2	—	**2**	—	**2**
shuttleM-un	14500	691	2	—	**169**	—	—
spect-un	267	22	2	2	**< 0.5**	2	**< 0.5**

machine learning. The main criteria for evaluating algorithms in this section is the accuracy of the test folds, while the training accuracy is given for completeness.

5.4.1 Comparison Against CART

We considered three tuning settings for our method to analyse the effect of restricting tuning options. This is mostly important for datasets for which computing the optimal decision tree is computationally expensive, and consequently limiting parameter values may be beneficial. The three settings are as follows:

1. MT-A: Fully exploit available parameters of our algorithm until depth four, i.e., $depth \in 1, 2, 3, 4$ and $feature\ node\ count \in \{depth, depth + 1, ..., 2^{depth} - 1\}$.

2. MT-R: We compute a reduced set of parameters by considering the depth and feature node count of the best decision tree produced by CART. The heuristically obtained tree provides an upper bound on the allowed parameter values for tuning, i.e., given a CART tree with depth d and number of feature nodes s, we tune with $depth \in 1, ..., d$ and $feature\ node\ count \in \{depth, ..., s\}$.

3. MT-F: Only a single parameter is set based on the CART tree. We fix the parameter values to match those produced by the best decision tree computed using CART. Note that this is not necessarily a hyper-tuning approach, but rather a method for selecting parameter values when computational time is limited.

Due to the number of considered datasets, we divided the tables of results into two parts: the upper (Table 5) and lower half (Table 6). The training and test accuracy is provided for the best tuned version of the methods, averaged across the five folds. The total time to perform tuning and retraining is given. CART was tuned using $depth \in [1, 2, 3, 4]$.

Our algorithm achieves consistently equal or better testing accuracy on the datasets regardless of the tuning strategy, with a few exceptions where the difference is marginal. This demonstrates that globally optimising trees captures the main features of the datasets more faithfully rather than locally optimising the tree in stages as in CART, even when not considering all possible parameter settings. The results are the best when all parameters are selected for tuning, and we observe a degradation as we limit the parameter options, but still remaining higher than CART in terms of accuracy.

The disadvantage of our algorithm when compared to CART is the computational time. While CART computes the trees in negligible time, the runtime to tune our algorithm may vary. In many cases, the runtime is under half a second, making it comparable to CART, but for some datasets the runtime may be in the range of hundreds or thousands of seconds. Nevertheless, the runtimes remain reasonably short and may be manageable for offline learning applications. For cases where training time is limited, our third most restricted parameter setting can serve as a possible heuristic for selecting the appropriate parameters for our method based on the characteristics of the tree produced by CART. If time is not critical, then naturally tuning with all parameters is the best option.

Table 5: (Upper half of the table) Comparison of our optimal classification trees and conventional heuristic approaches, CART and random forests. For each dataset, the number of instances, number of binary features, and number of classes are displayed. Training and testing accuracies using five-fold cross validation. Each method evaluated on exactly the same folds. Depth and size of the best trees given. Time is the total time to evaluate and train. Legend: MT-F, MT-R, and MT-A are our approach using fixed parameter based on CART, r parameter options based on CART, all parameter settings.

				Random Forest			CART				MT-F					MT-R					MT-A				
Name	$\|\mathcal{D}\|$	$\|\mathcal{F}\|$	$\|\mathcal{C}\|$	train	test	time	train	test	d	s	train	test	d	s	time	train	test	d	s	time	train	test	d	s	time
anneal	812	93	2	0.86	0.84	24	0.82	0.81	3	6	0.85	0.84	3	6	< 0.5	0.85	0.84	3	4	< 0.5	0.89	**0.87**	4	12	9
audiology	216	148	2	0.91	0.89	18	0.94	0.92	2	2	0.95	0.95	2	2	< 0.5	0.95	0.95	2	2	< 0.5	0.95	**0.95**	2	2	9
australian-credit	653	125	2	0.89	0.85	23	0.87	0.85	1	1	0.86	0.86	1	1	< 0.5	0.86	0.86	1	1	< 0.5	0.89	**0.87**	4	5	36
breast-wisconsin	683	120	2	0.97	0.96	22	0.96	0.93	4	13	0.99	0.95	4	13	1	0.98	0.97	3	4	16	0.98	**0.97**	3	4	16
diabetes	768	112	2	0.82	0.75	24	0.76	0.74	3	7	0.79	0.74	3	7	< 0.5	0.77	0.77	2	2	< 0.5	0.77	**0.77**	2	2	38
german-credit	1000	112	2	0.8	0.72	27	0.73	0.71	3	7	0.77	0.73	3	7	< 0.5	0.75	0.73	3	4	1	0.75	**0.73**	3	4	44
heart-cleveland	296	95	2	0.89	0.79	16	0.82	0.75	3	7	0.87	0.74	3	7	< 0.5	0.83	0.77	3	3	< 0.5	0.93	**0.79**	4	14	12
hepatitis	137	68	2	0.92	0.82	15	0.89	0.8	1	1	0.86	0.86	1	1	< 0.5	0.86	0.86	1	1	< 0.5	0.86	**0.86**	1	1	2
hypothyroid	3247	88	2	0.97	0.96	34	0.98	0.98	4	11	0.98	0.98	4	11	2	0.98	0.98	2	2	31	0.98	**0.98**	2	2	35
ionosphere	351	445	2	0.92	0.89	31	0.9	0.88	2	3	0.79	0.8	2	3	< 0.5	0.91	0.91	2	2	< 0.5	0.91	**0.91**	2	2	2701
kr-vs-kp	3196	73	2	0.9	0.89	38	0.82	0.82	4	7	0.95	0.95	4	7	2	0.95	0.95	4	7	16	0.95	**0.95**	4	7	27
letter	20000	224	2	0.97	0.97	534	0.97	0.97	4	15	0.99	0.99	4	15	289	0.99	0.99	4	6	5004	0.99	**0.99**	4	6	5980
lymph	148	68	2	0.9	0.83	15	0.87	0.81	2	3	0.66	0.67	2	3	< 0.5	0.84	0.84	2	2	< 0.5	0.92	**0.85**	3	6	3
mushroom	8124	119	2	0.96	0.96	74	0.94	0.94	4	8	1	1	4	8	2	1	1	3	5	84	1	**1**	3	5	88
pendigits	7494	216	2	0.97	0.97	143	0.97	0.97	4	14	1	1	4	14	69	1	1	4	7	1589	1	**1**	4	7	1915
primary-tumor	336	31	2	0.85	0.79	15	0.84	0.8	2	3	0.76	0.76	2	3	< 0.5	0.83	0.83	2	2	< 0.5	0.87	**0.85**	3	6	1
segment	2310	235	2	0.99	0.99	39	0.99	0.99	4	5	1	1	4	5	1	1	1	3	3	71	1	**1**	3	3	75
soybean	630	50	2	0.91	0.89	17	0.9	0.88	4	10	0.98	0.96	4	10	< 0.5	0.96	0.96	4	6	2	0.96	**0.96**	4	6	2
splice-1	3190	287	2	0.92	0.9	88	0.88	0.88	4	14	0.96	0.96	4	14	155	0.96	0.96	4	9	1756	0.96	**0.96**	4	9	2008
tic-tac-toe	958	27	2	0.8	0.79	18	0.75	0.73	4	13	0.87	0.79	4	13	< 0.5	0.86	0.82	4	11	2	0.86	**0.82**	4	11	2
vehicle	846	252	2	0.9	0.89	30	0.89	0.87	4	12	0.99	0.96	4	12	11	0.97	0.97	3	6	315	0.97	**0.97**	3	6	317
vote	435	48	2	0.97	0.95	15	0.97	0.95	3	7	0.98	0.93	3	7	< 0.5	0.96	0.96	1	1	< 0.5	0.96	**0.96**	1	1	2
yeast	1484	89	2	0.78	0.71	29	0.71	0.7	4	12	0.76	0.69	4	12	2	0.74	0.73	4	5	28	0.74	**0.73**	4	5	33
compas-binary	6907	12	2	0.66	0.66	38	0.65	0.65	4	15	0.67	0.67	4	15	< 0.5	0.67	0.67	3	4	2	0.67	**0.67**	3	4	2
fico-binary	10459	17	2	0.73	0.7	71	0.7	0.7	4	15	0.72	0.71	4	15	1	0.72	0.72	4	5	8	0.72	**0.72**	4	5	8
balance-scale	625	4	3	0.76	0.72	15	0.71	0.67	3	7	0.76	0.71	3	7	< 0.5	0.76	0.71	3	5	< 0.5	0.76	**0.71**	3	5	< 0.5
banknote	1372	16	2	0.89	0.89	17	0.84	0.84	4	5	0.92	0.92	4	5	< 0.5	0.92	0.92	4	5	< 0.5	0.95	**0.95**	4	9	< 0.5
bank_conv	4521	26	2	0.9	0.89	35	0.9	0.89	2	3	0.89	0.88	2	3	< 0.5	0.89	0.89	1	1	< 0.5	0.9	**0.9**	3	3	5
biodeg	1055	81	2	0.85	0.81	23	0.8	0.78	4	15	0.89	0.85	4	15	1	0.88	0.87	4	10	21	0.88	**0.87**	4	10	24
car_evaluation	1728	7	4	0.79	0.78	17	0.78	0.78	4	5	0.85	0.85	4	5	< 0.5	0.85	0.85	4	4	< 0.5	0.85	**0.85**	4	4	< 0.5
default_credit	30000	44	4	0.62	0.55	283	0.52	0.52	4	15	0.58	0.58	4	15	7	0.58	0.58	4	13	148	0.58	**0.58**	4	13	156
HTRU_2	17898	57	2	0.97	0.96	139	0.97	0.97	3	7	0.98	0.98	3	7	2	0.98	0.98	3	3	2	0.98	**0.98**	3	3	104
IndiansDiabetes	768	11	2	0.78	0.75	18	0.76	0.74	1	1	0.75	0.75	1	1	< 0.5	0.75	0.75	1	1	< 0.5	0.79	**0.77**	4	6	< 0.5
Ionosphere	351	143	2	0.94	0.91	21	0.91	0.89	4	7	0.97	0.9	4	7	2	0.95	0.94	4	4	27	0.95	**0.94**	4	4	34
iris	150	12	3	0.96	0.94	14	0.89	0.88	3	4	0.96	0.94	2	2	< 0.5	0.96	0.94	2	2	< 0.5	0.96	**0.94**	2	2	< 0.5
magic04	19020	79	2	0.79	0.76	232	0.77	0.77	4	15	0.82	0.82	4	15	7	0.82	0.82	4	12	149	0.82	**0.82**	4	12	155
messidor	1151	24	2	0.67	0.65	19	0.64	0.62	3	7	0.62	0.6	3	7	< 0.5	0.67	0.65	3	4	< 0.5	0.69	**0.66**	4	12	2
monk1_bin	124	15	2	0.88	0.81	15	0.81	0.77	3	4	0.91	0.91	3	4	< 0.5	0.91	0.91	3	4	< 0.5	1	**1**	4	8	< 0.5
monk2_bin	169	15	2	0.77	0.61	15	0.68	0.59	1	1	0.63	0.59	1	1	< 0.5	0.63	0.59	1	1	< 0.5	0.77	**0.67**	3	6	< 0.5
monk3_bin	122	15	2	0.94	0.91	15	0.9	0.88	2	3	0.78	0.78	2	3	< 0.5	0.93	0.93	2	2	< 0.5	0.93	**0.93**	2	2	< 0.5
seismic_bumps	2584	10	2	0.93	0.93	22	0.93	0.93	1	1	0.93	0.93	1	1	< 0.5	0.93	0.93	1	1	< 0.5	0.93	**0.93**	1	1	< 0.5
spambase	4601	132	2	0.9	0.88	81	0.85	0.84	4	14	0.92	0.91	4	14	8	0.91	0.91	4	7	172	0.91	**0.91**	4	7	186
Statlog_satellite	4435	385	6	0.74	0.71	194	0.55	0.54	4	15	0.77	0.76	4	15	270	0.76	0.76	4	12	4839	0.76	**0.76**	4	12	5148
Statlog_shuttle	43500	181	7	0.9	0.9	1273	0.93	0.93	4	10	1	1	4	10	49	1	1	4	6	2033	1	**1**	4	6	2125
tic-tac-toe_bin	958	18	2	0.81	0.8	17	0.75	0.73	4	13	0.87	0.79	4	13	< 0.5	0.86	0.82	4	11	1	0.86	**0.82**	4	11	1
winequality-red	1599	17	6	0.62	0.58	21	0.58	0.57	3	7	0.6	0.57	3	7	< 0.5	0.58	0.58	3	3	< 0.5	0.62	**0.6**	4	11	1
wine	178	32	3	0.98	0.96	15	0.88	0.84	3	4	1	0.97	4	6	< 0.5	0.99	0.98	4	5	< 0.5	0.99	**0.98**	4	5	< 0.5

5.4.2 Comparison Against Random Forests

The main comparison is done against decision tree algorithms, but for completeness we compare optimal decision trees with tuned random forests using the same sklearn Python package. A forest of trees is typically more accurate than a single decision tree, but the resulting model is less concise and more difficult for human interpretation. The forests were tuned by varying the number of trees in the forest from $[10, 50, 100]$, selecting the maximum depth from $[no\_limit, 1, 2, 4]$, and considering a subset of the features at each step to evaluate with respect to $[|\mathcal{F}|, \frac{1}{2}|\mathcal{F}|, \sqrt{|\mathcal{F}|}, log_2(|\mathcal{F}|)]$, where $\mathcal{F}$ is the set of features.

Table 6: (Lower half of the table) Comparison of our optimal classification trees and conventional heuristic approaches, CART and random forests. For each dataset, the number of instances, number of binary features, and number of classes are displayed. Training and testing accuracies using five-fold cross validation. Each method evaluated on exactly the same folds. Depth and size of the best trees given. Time is the total time to evaluate and train. Legend: MT-F, MT-R, and MT-A are our approach using fixed parameter based on CART, r parameter options based on CART, all parameter settings.

				Random Forest			CART				MT-F					MT-R					MT-A				
Name	$\|\mathcal{D}\|$	$\|\mathcal{F}\|$	$\|\mathcal{C}\|$	train	test	time	train	test	d	s	train	test	d	s	time	train	test	d	s	time	train	test	d	s	time
appendicitis-un	106	530	2	0.86	0.8	17	0.87	0.79	1	1	0.84	0.8	1	1	< 0.5	0.84	0.8	1	1	< 0.5	0.86	**0.84**	2	2	146
australian-un	690	1163	2	0.82	0.77	47	0.87	0.84	1	1	0.86	0.86	1	1	1	0.86	0.86	1	1	< 0.5	0.86	**0.86**	1	1	11102
backache-un	180	475	2	0.9	0.86	19	0.9	0.83	1	1	0.88	0.82	1	1	< 0.5	0.88	0.82	1	1	< 0.5	0.86	**0.86**	2	3	325
cancer-un	683	89	2	0.96	0.94	19	0.94	0.93	3	6	0.98	0.93	3	6	1	0.97	0.94	3	6	7	0.97	**0.94**	3	6	8
car-un	1728	21	2	0.86	0.85	22	0.83	0.83	4	5	0.91	0.91	4	5	< 0.5	0.91	0.91	4	5	< 0.5	0.91	**0.91**	4	5	2
cleve-un	303	395	2	0.87	0.79	18	0.83	0.79	3	7	0.87	0.81	3	7	< 0.5	0.86	0.83	3	5	1	0.86	**0.83**	3	5	182
colic-un	368	415	2	0.85	0.79	21	0.87	0.84	2	3	0.75	0.73	2	3	< 0.5	0.86	0.85	2	2	< 0.5	0.86	**0.85**	2	2	526
corral-un	160	6	2	0.93	0.91	13	0.84	0.81	4	11	1	1	4	11	< 0.5	1	1	4	7	< 0.5	1	**1**	4	7	< 0.5
haberman-un	306	92	2	0.8	0.73	15	0.76	0.71	1	1	0.75	0.72	1	1	< 0.5	0.75	0.72	1	1	< 0.5	0.75	**0.72**	1	1	5
heart-statlog-un	270	381	2	0.86	0.78	17	0.83	0.77	3	7	0.88	0.81	3	7	< 0.5	0.87	0.86	3	5	1	0.87	**0.86**	3	5	145
hepatitis-un	155	361	2	0.88	0.81	16	0.88	0.82	1	1	0.85	0.84	1	1	< 0.5	0.85	0.84	1	1	< 0.5	0.85	**0.84**	1	1	191
house-votes-84-un	435	16	2	0.97	0.95	15	0.96	0.95	1	1	0.96	0.96	1	1	< 0.5	0.96	0.96	1	1	< 0.5	0.96	**0.96**	1	1	< 0.5
hungarian-un	294	330	2	0.85	0.78	17	0.85	0.78	3	7	0.77	0.76	2	3	< 0.5	0.82	0.81	1	1	< 0.5	0.82	**0.81**	1	1	170
irish-un	500	112	2	0.95	0.95	16	0.96	0.96	3	3	0.95	0.96	3	3	< 0.5	0.99	0.99	2	2	< 0.5	1	**1**	4	7	< 0.5
mouse-un	70	45	2	0.97	0.91	13	0.94	0.84	3	5	0.97	0.9	3	5	< 0.5	0.97	0.91	3	3	< 0.5	0.97	**0.91**	3	3	< 0.5
mux6-un	128	6	2	0.86	0.81	13	0.73	0.61	4	13	1	1	4	13	< 0.5	1	1	3	7	< 0.5	1	**1**	3	7	< 0.5
new-throid-un	215	334	3	0.78	0.71	18	0.76	0.7	2	2	0.8	0.74	4	7	13	0.8	0.74	4	7	167	0.8	**0.74**	4	7	186
promoters-un	106	334	2	0.96	0.85	16	0.91	0.75	3	7	0.99	0.79	3	7	< 0.5	0.93	0.79	3	4	2	0.96	**0.83**	4	5	345
shuttleM-un	14500	691	2	0.89	0.89	2004	0.92	0.92	4	9	1	1	4	9	154	1	1	4	4	5169	1	**1**	4	4	5592
spect-un	267	22	2	0.85	0.81	14	0.81	0.78	1	1	0.79	0.79	1	1	< 0.5	0.79	0.79	1	1	< 0.5	0.84	**0.81**	4	4	< 0.5
appendicitis-un-r	106	40	2	0.92	0.88	15	0.93	0.88	2	2	0.91	0.88	1	1	< 0.5	0.91	0.88	1	1	< 0.5	0.92	**0.91**	3	5	< 0.5
australian-un-r	690	23	2	0.96	0.96	16	0.97	0.97	4	8	0.99	0.99	4	8	< 0.5	0.99	0.99	3	6	< 0.5	0.99	**0.99**	3	6	< 0.5
backache-un-r	180	15	2	0.96	0.94	14	0.96	0.94	3	5	0.95	0.97	3	5	< 0.5	0.98	0.97	3	4	< 0.5	0.98	**0.97**	3	4	< 0.5
cancer-un-r	449	9	2	0.96	0.94	15	0.92	0.9	4	11	0.98	0.95	4	11	< 0.5	0.98	0.96	4	7	< 0.5	0.98	**0.96**	4	7	< 0.5
car-un-r	1728	8	2	0.9	0.9	19	0.87	0.86	4	8	0.97	0.97	4	8	< 0.5	0.97	0.97	4	7	< 0.5	0.97	**0.97**	4	7	< 0.5
cleve-un-r	302	6	2	0.96	0.96	15	0.92	0.92	4	7	1	1	4	7	< 0.5	1	1	4	6	< 0.5	1	**1**	4	6	< 0.5
colic-un-r	357	10	2	0.96	0.95	16	0.96	0.95	4	8	0.99	0.99	3	6	< 0.5	0.99	0.99	3	5	< 0.5	0.99	**0.99**	3	5	< 0.5
corral-un-r	160	6	2	0.93	0.9	15	0.84	0.81	4	11	1	1	4	11	< 0.5	1	1	4	7	< 0.5	1	**1**	4	7	< 0.5
haberman-un-r	289	15	2	0.96	0.94	15	0.96	0.94	4	8	0.98	0.97	4	8	< 0.5	0.98	0.97	4	7	< 0.5	0.98	**0.97**	4	7	< 0.5
heart-statlog-un-r	270	8	2	0.93	0.92	15	0.9	0.89	4	12	0.99	0.97	4	12	< 0.5	0.98	0.98	4	6	< 0.5	0.98	**0.98**	4	6	< 0.5
hepatitis-un-r	155	7	2	0.97	0.97	14	0.99	0.99	2	3	0.97	0.97	2	3	< 0.5	0.99	0.99	2	2	< 0.5	0.99	**0.99**	2	2	< 0.5
hungarian-un-r	293	10	2	0.98	0.96	16	0.97	0.96	3	5	0.98	0.96	3	5	< 0.5	0.98	0.98	3	3	< 0.5	0.98	**0.98**	3	3	< 0.5
irish-un-r	500	112	2	0.95	0.95	18	0.96	0.96	3	3	0.95	0.96	3	3	< 0.5	0.99	0.99	2	2	< 0.5	1	**1**	4	7	< 0.5
mouse-un-r	70	45	2	0.97	0.91	15	0.94	0.84	3	5	0.97	0.9	3	5	< 0.5	0.97	0.91	3	3	< 0.5	0.97	**0.91**	3	3	< 0.5
mux6-un-r	128	6	2	0.86	0.82	16	0.73	0.62	4	13	1	1	4	13	< 0.5	1	1	3	7	< 0.5	1	**1**	3	7	< 0.5
promoters-un-r	106	6	2	0.95	0.91	14	0.9	0.82	4	9	0.99	0.92	4	9	< 0.5	0.97	0.92	4	6	< 0.5	0.97	**0.92**	4	6	< 0.5
shuttleM-un-r	14500	5	2	0.92	0.92	45	0.9	0.9	4	8	1	1	4	8	< 0.5	1	1	3	3	< 0.5	1	**1**	3	3	< 0.5
spect-un-r	228	9	2	0.98	0.98	14	0.98	0.97	1	1	0.97	0.97	1	1	< 0.5	0.97	0.97	1	1	< 0.5	0.97	**0.97**	1	1	< 0.5

We show the results in Tables 5 and 6. Random forests tend to outperform CART in terms of accuracy, but not on every dataset. However, optimal decision trees provide higher accuracy on almost all datasets despite the simplicity of the their resulting model. This highlights that fully optimising a method may be more beneficial than using more complex but less optimised models. The runtime to tune random forests differs among the datasets, but it is clear that it is no longer negligible as for CART. Our approach can achieve lower or comparative training times when compared to random forests, although for several datasets the runtimes may be considerably higher.

6. Conclusion

We presented MurTree, a framework for computing optimal decision trees, i.e., decision trees that achieve the best representation of the data in terms of minimising the number of

misclassifications. The framework is based on dynamic programming and search. Our novel techniques exploit decision tree properties to provide orders of magnitude speed-ups when compared to the state-of-the-art. The conducted experimental study shows that optimal decision trees are highly desirable as their out-of-sample accuracy is greater than decision trees and random forests obtained using conventional learning algorithms, while providing concise and interpretable models. Traditional heuristic algorithms are typically faster, but we show that for the majority of the datasets tested our approach can compute optimal trees within seconds or minutes.

Considering novel metrics for evaluating optimality to improve the ability to generalise better on unseen data may be a direction for future work. In particular, pruning techniques from heuristic approaches, that are typically applied as a post-processing step, may be incorporated directly in the optimal decision tree objective. Analysing the effect of supervised discretisation algorithms for binarising the datasets may lead to additional insights. Lastly, our efforts were mainly focussed on trees of depth at most five, and extending our techniques to handle much deeper trees, such as depth ten, would be of interest for particular applications.

References

Sina Aghaei, Mohammad Javad Azizi, and Phebe Vayanos. Learning optimal and fair decision trees for non-discriminative decision-making. In *Proceedings of AAAI-19*, volume 33, pages 1418–1426, 2019.

Gaël Aglin, Siegfried Nijssen, and Pierre Schaus. Learning optimal decision trees using caching branch-and-bound search. In *Proceedings of AAAA-20*, 2020.

Pranav Ashok, Mathias Jackermeier, Pushpak Jagtap, Jan Křetínskỳ, Maximilian Weininger, and Majid Zamani. dtcontrol: decision tree learning algorithms for controller representation. *arXiv preprint arXiv:2002.04991*, 2020.

Florent Avellaneda. Efficient inference of optimal decision trees. In *Proceedings of AAAI-20*, 2020.

Osbert Bastani, Yewen Pu, and Armando Solar-Lezama. Verifiable reinforcement learning via policy extraction. In *Advances in Neural Information Processing Systems*, pages 2494–2504, 2018.

Dimitris Bertsimas and Jack Dunn. Optimal classification trees. *Machine Learning*, 106(7): 1039–1082, 2017.

Dimitris Bertsimas and Romy Shioda. Classification and regression via integer optimization. *Operations Research*, 55(2):252–271, 2007.

Leo Breiman, JH Friedman, RA Olshen, and CJ Stone. Classification and regression trees. *Cole Statistics/Probability Series*, 1984.

Usama M. Fayyad and Keki B. Irani. Multi-interval discretization of continuous-valued attributes for classification learning. In Ruzena Bajcsy, editor, *Proceedings of IJCAI'93*, 1993.

Michael R Garey. Optimal binary identification procedures. *SIAM Journal on Applied Mathematics*, 23(2):173–186, 1972.

Xiyang Hu, Cynthia Rudin, and Margo Seltzer. Optimal sparse decision trees. In *Advances in Neural Information Processing Systems*, pages 7265–7273, 2019.

HyunJi Kim. Package discretization in cran-r. `https://CRAN.R-project.org/package=discretization`, 2015. [Online; accessed 21-May-2020].

Hyafil Laurent and Ronald L Rivest. Constructing optimal binary decision trees is NP-complete. *Information Processing Letters*, 5(1):15–17, 1976.

Nina Narodytska, Alexey Ignatiev, Filipe Pereira, and Joao Marques-Silva. Learning optimal decision trees with SAT. In *Proceedings of IJCAI-18*, pages 1362–1368, 2018.

Siegfried Nijssen and Elisa Fromont. Mining optimal decision trees from itemset lattices. In *Proceedings of SIGKDD-07*, pages 530–539, 2007.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

Ross Quinlan. *C4.5: Programs for Machine Learning*. Kaufmann, 1993.

Hélene Verhaeghe, Siegfried Nijssen, Gilles Pesant, Claude-Guy Quimper, and Pierre Schaus. Learning optimal decision trees using constraint programming. In *Proceedings of CP-19*, 2019.

Sicco Verwer and Yingqian Zhang. Learning decision trees with flexible constraints and objectives using integer optimization. In *Proceedings of CPAIOR-17*, pages 94–103. Springer, 2017.

Sicco Verwer and Yingqian Zhang. Learning optimal classification trees using a binary linear program formulation. In *Proceedings of AAAI-19*, volume 33, pages 1625–1632, 2019.