Optimal Decision Trees

Algorithms and Applications

Jacobus G. M. van der Linden

Optimal Decision Trees

Algorithms and Applications

Optimal Decision Trees

Algorithms and Applications

Dissertation

for the purpose of obtaining the degree of doctor
at Delft University of Technology,
by the authority of the Rector Magnificus,
Prof.dr.ir. H. Bijl,
chair of the Board for Doctorates,
to be defended publicly on
Wednesday June 10, 2026 at 17:30

by

Jacobus Gerrit Mattheus van der Linden

This dissertation has been approved by the (co)promotors.

Composition of the doctoral committee:

Rector Magnificus,	Chair
Prof.dr. M. M. de Weerdt,	Delft University of Technology, promotor
Dr. E. Demirović,	Delft University of Technology, copromotor
Independent members:	
Prof.dr. M. D. Romero Morales,	Copenhagen Business School
Prof.dr. J. C. Beck,	University of Toronto
Prof.dr. M. van Leeuwen,	Leiden University
Prof.dr.ir. M. J. T. Reinders,	Delft University of Technology
Dr.ir. L. J. J. van Iersel,	Delft University of Technology
Reserve member:	
Prof.dr. F.A. Oliehoek,	Delft University of Technology

Keywords:	Decision trees, dynamic programming, machine learning, survival analysis, Rashomon sets, optimization
Printed by:	Proefschriftspecialist
Cover inspired by:	Kruszewski (1996), "The Botanical Beauty of Random Binary Trees: A Method for the Synthetic Imagery of Botanical Trees"
Copyright:	2026 Jacobus G. M. van der Linden
ISBN:	978-94-6518-342-8
Code appendix:	`https://doi.org/10.4121/3584f9cd-cf7b-478c-960f-3b2ef47d1427`

An electronic version of the dissertation is available at `http://repository.tudelft.nl/`.

Blessed is the one who finds wisdom,
and the one who gets understanding,
for the gain from her is better than gain from silver
and her profit better than gold.
She is more precious than jewels,
and nothing you desire can compare with her.
Long life is in her right hand;
in her left hand are riches and honor.
Her ways are ways of pleasantness,
and all her paths are peace.
She is a tree of life to those who lay hold of her;
those who hold her fast are called blessed.

Proverbs 3:13-18

Contents

Summary **xiii**

Samenvatting **xv**

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Challenges . . . 3
1.3 Research Questions . . . 6
1.3.1 Objectives and Constraints . . . 6
1.3.2 Comparing Optimal and Greedy Decision Trees . . . 7
1.3.3 Scalability with Numerical Features . . . 8
1.3.4 Enumerating the Whole Rashomon Set . . . 9
1.3.5 Comparing Model- and Search-Based Approaches . . . 9
1.4 Summary of Contributions . . . 10

2 Background **13**
2.1 Greedy Learning . . . 13
2.2 Complexity . . . 15
2.3 Search-Based Solving . . . 17
2.3.1 Dynamic Programming . . . 17
2.3.2 Caching . . . 18
2.3.3 Bound-Based Pruning . . . 19
2.3.4 Search Strategy . . . 22
2.3.5 Depth-Two Case . . . 26
2.3.6 Anytime Performance . . . 27
2.3.7 Continuous Features . . . 27
2.3.8 Comparison . . . 28
2.4 Model-Based Solving . . . 31
2.4.1 Mixed-Integer Programming . . . 32
2.4.2 Continuous Optimization . . . 36
2.4.3 Constraint Programming . . . 36
2.4.4 Boolean Satisfiability . . . 37
2.5 Applications . . . 39

3 Fair and Optimal Decision Trees **43**

3.1 Introduction . . . 43
3.2 Related Work . . . 45
3.3 Preliminaries . . . 46
3.4 Method Description . . . 48
3.5 Experimental Results . . . 52
3.6 Conclusion . . . 55
Appendices . . . 56
3.A Complexity Analysis . . . 56
3.B Non-Complete Trees . . . 56
3.C Dataset Details . . . 56
3.D Test Evaluation . . . 57

4 A General DP Approach: Theory and Applications 61
4.1 Introduction . . . 61
4.2 Related Work . . . 62
4.3 Preliminaries . . . 64
4.4 Framework for Separable Objectives . . . 65
4.4.1 Problem Definition and Notation . . . 65
4.4.2 Separability . . . 66
4.4.3 Dynamic Programming Formulation . . . 68
4.4.4 Examples of Separable Optimization Tasks . . . 68
4.4.5 Comparison to Previous Theory and Methods . . . 72
4.5 Experiments . . . 73
4.5.1 Cost-Sensitive Classification . . . 73
4.5.2 Prescriptive Policy Generation . . . 79
4.5.3 Nonlinear Classification Metrics . . . 82
4.5.4 Group Fairness . . . 83
4.5.5 Classification Accuracy . . . 85
4.5.6 Summary of the Results . . . 87
4.6 Conclusion . . . 88
Appendices . . . 89
4.A Proofs . . . 89
4.B Pseudo-Code and Implementation . . . 95
4.C Related Work for the Application Domains . . . 102

5 Survival Analysis 107
5.1 Introduction . . . 107
5.2 Related Work . . . 109
5.3 Preliminaries . . . 109
5.4 Method . . . 111
5.4.1 Loss Function . . . 111
5.4.2 Dynamic Programming Approach . . . 112
5.4.3 Trees of Depth Two . . . 113
5.5 Experiments . . . 114
5.5.1 Experiment Setup . . . 114
5.5.2 Survival Metrics . . . 115

5.5.3 Scalability . . . 117
5.5.4 Out-of-Sample Results . . . 118
5.6 Conclusion . . . 119
Appendices . . . 120
5.A Loss Function . . . 120
5.B Depth-Two Solver . . . 121
5.C Distributions in Ground Truth Trees . . . 122
5.D Extended Experiment Results . . . 122

6 Piecewise Constant and Linear Regression 125
6.1 Introduction . . . 125
6.2 Related Work . . . 127
6.3 Preliminaries . . . 128
6.4 Piecewise Constant Regression Trees . . . 129
6.5 Piecewise Linear Regression Trees . . . 131
6.5.1 Multiple Linear Regression . . . 131
6.5.2 Simple Linear Regression . . . 132
6.6 Experiments . . . 133
6.6.1 Experiment Setup . . . 133
6.6.2 Scalability . . . 135
6.6.3 Out-of-Sample Performance . . . 137
6.7 Conclusion . . . 139
Appendices . . . 140
6.A Pseudocode . . . 140
6.B Simple Linear Regression Derivation . . . 140
6.C Training Results . . . 142

7 Empirical Analysis of Optimal and Greedy Decision Trees 145
7.1 Introduction . . . 146
7.2 The Optimization Objective for ODTs . . . 148
7.2.1 Greedy Proxies for Accuracy . . . 148
7.2.2 Analysis of ODT Objectives . . . 149
7.2.3 Novel Non-Concave Objectives . . . 150
7.2.4 Experiments . . . 151
7.2.5 Discussion . . . 155
7.3 Tuning the Complexity of ODTs . . . 156
7.3.1 Tuning Approaches . . . 156
7.3.2 Experiments . . . 156
7.3.3 Conclusion . . . 158
7.4 Comparing Optimal and Greedy Decision Trees . . . 159
7.4.1 Previous Comparisons . . . 160
7.4.2 Training CART . . . 162
7.4.3 Accuracy-Interpretability Trade-Off . . . 164
7.4.4 Experiments on Literature Claims . . . 165
7.4.5 Scalability of Optimal Decision Trees . . . 173
7.5 Conclusion . . . 173

Appendices . . . 175
7.A Splitting and Pruning Criteria as ODT Objectives . . . 175
7.B ODT Tuning Approaches . . . 177
7.C Previous Comparisons between Optimal and CART . . . 180
7.D Additional Experiments . . . 182
7.E Datasets . . . 184

8 Scalability for Optimization with Continuous Features 191
8.1 Introduction . . . 191
8.2 Preliminaries . . . 192
8.3 The ConTree Algorithm . . . 193
8.3.1 Pruning Techniques . . . 194
8.3.2 General Recursive Case . . . 197
8.3.3 Depth-Two Subroutine . . . 199
8.3.4 Optimality Gap . . . 199
8.3.5 Comparison to Previous Bounding Methods . . . 199
8.4 Experiments . . . 200
8.5 Conclusion . . . 203
Appendices . . . 205
8.A Data Preprocessing . . . 205
8.B Results for Larger Depth Limits . . . 206
8.C Memory Usage Results . . . 206
8.D Out-of-Sample Results . . . 207

9 Enumerating the Rashomon Set of Close-to-Optimal Trees 211
9.1 Introduction . . . 212
9.2 Related Work . . . 213
9.3 Preliminaries . . . 214
9.4 In-Order Rashomon Set Calculation . . . 215
9.4.1 High-level Idea . . . 215
9.4.2 Algorithm . . . 216
9.4.3 Comparison to the State of the Art . . . 218
9.5 Experimental Evaluation . . . 219
9.5.1 Runtime Performance . . . 220
9.5.2 Variable Importance Analysis . . . 221
9.5.3 Evaluating Other Objectives . . . 223
9.6 Conclusion . . . 224
Appendices . . . 225
9.A Detailed Method Description . . . 225
9.B Experiment Details . . . 231

10 Conclusion 249
10.1 Answers to the Research Questions . . . 249
10.2 Model- and Search-Based Approaches . . . 253
10.3 Limitations and Future Work . . . 254

Bibliography **257**

Acknowledgements **277**

Curriculum Vitæ **281**

List of Publications **283**

Summary

In recent years, artificial intelligence has increasingly permeated society, including applications in high-stakes domains that may have a significant and even harmful impact on people. For these domains, it is essential to have *comprehensible* and *reliable* models. However, many of the most commonly used models, such as neural networks, are black-box models, i.e., inherently opaque and thus hard to trust.

Therefore, this dissertation explores decision trees as a human-interpretable alternative. Specifically, we focus on *optimal decision trees*, since the traditional greedy heuristics often learn decision trees that are too large to interpret easily, whereas optimal decision trees provably optimize the objective of a learning problem (e.g., accuracy) given a constraint on model size. The compactness and transparency of these optimal models contribute to their interpretability, while in many cases they achieve accuracy comparable to that of much more complex black-box models. In addition, with many optimal learning methods it is easier to specify a learning task other than the default of maximizing classification accuracy.

However, since learning optimal decision trees is an NP-hard problem, *scalability* is a major concern. State-of-the-art model-based approaches, such as mixed-integer programming and Boolean satisfiability, often do not scale beyond a few hundred samples and shallow depth limits. On the other hand, search-based approaches, such as *dynamic programming*, scale much better by exploiting the *separable* tree structure using specialized algorithms to solve subtrees as independent (and repeated) subproblems. But these methods are mostly limited to classification, while their scalability depends on a coarse binarization of the numerical data.

Therefore, in this thesis, we study optimal decision trees across four challenges: (i) efficiently optimizing learning tasks other than classification, (ii) improving our understanding of the differences between optimal and traditional greedy decision trees, (iii) efficiently handling data with numerical attributes; and (iv) efficiently finding the whole Rashomon set: the set of all (near-)optimal decision trees.

In the first challenge we address improving scalability for optimizing decision trees for *a variety of learning tasks* other than classification. We build on existing dynamic programming approaches for their scalability, and because this scalability depends on the property that subproblems are separable, we study and formalize the precise *necessary and sufficient conditions for separability*. These conditions turn out to be broader than previously assumed, and we use these new insights to improve the scalability of optimal decision trees for a variety of learning tasks, such as classification under fairness constraints, survival analysis, regression, prescriptive policy learning, and cost-sensitive classification. We also show how different optimization techniques, such as branch-and-bound and a specialized subroutine for depth-two trees, can be adapted and applied across several of these tasks. Together, these techniques

improve scalability by one or more orders of magnitude over the state of the art for multiple learning tasks.

The second challenge is to improve our understanding of *the differences between optimal and greedy heuristic methods* for learning decision trees because previous comparisons were severely limited by lack of scalability. We empirically study the effect of the loss function and hyperparameter tuning of optimal decision trees and conclude that the heuristic and optimal approach are inherently different, for example, concave loss functions, which are required for many heuristics, do not work well for optimal methods. Previous comparisons also resulted in divergent and contradicting claims about optimal methods, and therefore, we empirically test six such claims from the literature. We confirm the most important one: when measuring both size (as a proxy of interpretability) and accuracy, optimal decision trees clearly dominate trees learned by greedy heuristics. We also refute several claims, including the claim that optimal methods are more prone to overfitting, and the claim that the differences between the two methods diminish with more data. For both claims, we find evidence suggesting the opposite.

The third challenge is to improve scalability when the data contains *numerical attributes*, without falling back to a coarse binarization as most dynamic programming approaches do. Therefore, we develop new algorithmic techniques that exploit the properties of numerical attributes. We show how reasoning about subproblem similarity makes it possible to prove that large parts of the search space can be pruned, while the ordering among numeric values makes it possible to compute aspects of the problem incrementally. Together, these techniques result in scalability improvements of one to two orders of magnitude compared to the state of the art.

For the fourth challenge, we go beyond learning a single optimal model. Instead, we focus on the set of all decision trees within a small percentage of the optimal solution, the so-called *Rashomon set*. The Rashomon set can be useful for data analysis, for example in determining which attributes have the greatest impact on predictions, or in finding better counterfactual explanations. Our contribution is a more efficient, anytime, sorted enumeration of all decision trees in the Rashomon set using lazy evaluation, as well as adapting and applying existing algorithmic techniques for optimal decision trees to compute the Rashomon set. Here too, our method improves scalability by one or more orders of magnitude compared to the previous best approach.

In summary, this dissertation improves scalability by several orders of magnitude, broadens the range of possible applications, and deepens both the theoretical and empirical understanding of optimal decision trees and decision tree Rashomon sets. Therefore, we can now recommend the use of optimal decision trees as human interpretable models for a large variety of real-world data science problems.

Samenvatting

Kunstmatige intelligentie heeft een steeds grotere impact op de samenleving en wordt ook meer en meer ingezet voor toepassingen die significante of zelfs schadelijke impact kunnen hebben op mensen. Zulke toepassingen vereisen modellen die we kunnen *begrijpen* en *vertrouwen*, maar helaas zijn veel van de meestgebruikte modellen, zoals neurale netwerken, inherent ondoorzichtig en dus moeilijk te vertrouwen.

Daarom onderzoeken we in deze dissertatie *beslisbomen* als een alternatief die wel begrijpbaar is voor mensen. De focus ligt op *optimale beslisbomen*, omdat de traditionele niet-optimale heuristieken veelal beslisbomen leren die te groot zijn om gemakkelijk te interpreteren, terwijl optimale beslisbomen juist aantoonbaar de doelfunctie van een leerprobleem (bijv. nauwkeurigheid) optimaliseren voor een gegeven beperking op de grootte. De compactheid en de transparantie van deze optimale modellen draagt dus bij aan hun begrijpbaarheid, terwijl ze in veel gevallen in nauwkeurigheid weinig afdoen aan veel complexere en niet-doorzichtige modellen. Daarnaast is het bij veel optimale leermethoden makkelijker om ook andere leertaken te optimaliseren naast het maximaliseren van nauwkeurigheid.

Het leren van optimale beslisbomen is echter een NP-moeilijk probleem, en dus is *schaalbaarheid* een belangrijk aandachtspunt. De nieuwste model-gebaseerde methoden, zoals gemengd geheeltallig programmeren (mixed-integer programming) en Boolean satisfiability (SAT), schalen vaak niet voorbij enkele honderden datapunten en een zeer beperkt dieptelimiet. Zoekmethoden, zoals *dynamisch programmeren*, daarentegen, schalen veel beter omdat ze de *scheidbaarheid* van een boom uitbuiten door speciale algoritmes die subbomen als onafhankelijke (en herhaalde) problemen oplossen. Deze methoden echter zijn meestal beperkt tot classificatie, en de schaalbaarheid wordt behaald door een grove binarisatie van numerieke data.

Daarom onderzoeken we in deze dissertatie vier uitdagingen omtrent het leren van optimale beslisbomen: (i) het efficiënt optimaliseren van verschillende leertaken naast classificatie, (ii) het vergroten van ons begrip van de verschillen tussen optimale en traditionele beslisbomen, (iii) het efficiënt omgaan met numerieke data, en (iv) het efficiënt vinden van de volledige Rashomon verzameling: de verzameling van alle beslisbomen die (bijna) optimaal zijn.

Voor de eerste uitdaging verbeteren we de schaalbaarheid van het leren van optimale beslisbomen voor *verschillende leertaken* naast classificatie. We bouwen voort op een bestaande dynamisch-programmeren techniek vanwege haar schaalbaarheid. Omdat deze schaalbaarheid afhangt van de eigenschap dat subproblemen scheidbaar zijn, onderzoeken en formuleren we de precieze *noodzakelijke en voldoende voorwaarden voor scheidbaarheid*. Deze voorwaarden blijken breder te zijn dan voorheen aangenomen, en we gebruiken dit nieuw inzicht om de schaalbaarheid te verbeteren voor verscheidene leertaken, zoals classificatie onderhevig aan eerlijkheidsvoorwaarden, overlevingsanalyse, regressie, het leren van voorschrijvende

beleidsfuncties, en kost-sensitieve classificatie. We laten ook zien hoe verschillende optimalisatie technieken, zoals branch-and-bound en een speciale subroutine voor bomen van diepte twee, aangepast kunnen worden voor meerdere van deze leertaken. Gezamenlijk verbeteren deze technieken de schaalbaarheid van meerdere leertaken met één of meerdere ordes van grootte ten opzichte van de stand van de techniek.

De tweede uitdaging is het beter begrijpen van *de verschillen tussen optimale en heuristische methoden* om beslisbomen te leren omdat voorgaande vergelijkingen zeer beperkt waren vanwege gebrek aan schaalbaarheid. We bestuderen empirisch het effect van de leerfunctie en de hyperparameterafstemming van optimale beslisbomen en concluderen dat optimale methoden en heuristieken inherent verschillend zijn. Bijv., concave leerfuncties, die vereist zijn voor veel heuristieken, werken juist niet goed voor optimale methoden. Voorgaande vergelijkingen resulteerden ook in niet-éénduidige en zelfs tegengestelde claims over optimale methoden, en daarom testen we empirisch zes van zulke claims uit de literatuur. We bevestigen de belangrijkste claim: als zowel modelgrootte (als maatstaf van interpreteerbaarheid) en nauwkeurigheid gemeten wordt, dan zijn optimale beslisbomen overduidelijk beter dan bomen geleerd met een heuristiek. We weerleggen ook meerdere claims, waaronder de claim dat optimale methoden gevoeliger zouden zijn voor overaanpassing (overfitting), en de claim dat de verschillen tussen de twee methoden minder worden met meer data. Wij zien voor beide claims juist reden het tegenovergestelde te geloven.

De derde uitdaging is het verbeteren van de schaalbaarheid als de data uit *numerieke attributen* bestaat, zonder terug te vallen op een grove binarisatie zoals de meeste dynamisch-programeren methoden doen. Daarom ontwikkelen we nieuwe algoritmische technieken die de eigenschappen van numerieke attributen uitbuiten. We laten zien hoe redeneren over soortgelijke subproblemen helpt om aan te tonen dat grote delen van de zoekruimte overgeslagen kunnen worden, en hoe de rangschikking van getallen helpt om bepaalde aspecten van het probleem incrementeel te berekenen. Gezamenlijk zorgen deze technieken voor een verbetering in schaalbaarheid van één of twee ordes van grootte ten opzichte van vorige methoden.

Voor de vierde uitdaging gaan we verder dan het leren van slechts één optimaal model. In plaats daarvan berekenen we de verzameling van alle beslisbomen die binnen een klein percentage van de optimale oplossing zitten, de zogeheten *Rashomon verzameling*. De Rashomon verzameling kan helpen in data-analyse, zoals in het bepalen welke attributen de grootste impact op voorspellingen hebben, of in het vinden van betere tegenfeitelijke verklaringen (counterfactual explanations). Onze bijdrage is het efficiënter anytime gesorteerd opsommen van alle beslisbomen in de Rashomon verzameling door middel van luie evaluatie, en het aanpassen en toepassen van bestaande algoritmische technieken voor optimale beslisbomen voor het zoeken van de Rashomon verzameling. Ook hier verbetert onze methode de schaalbaarheid met één of meerdere ordes van grootte ten opzichte van de vorige beste methode.

Deze dissertatie verbetert dus de schaalbaarheid met meerdere ordes van grootte, verbreedt mogelijke toepassingen, en vergroot ons theoretisch en empirisch begrip van optimale beslisbomen en de Rashomon verzameling van beslisbomen. Daarom kunnen we nu het gebruik van optimale beslisbomen als begrijpbaar alternatief aanraden voor een verscheidenheid van praktijktoepassingen in data science.

1

Introduction

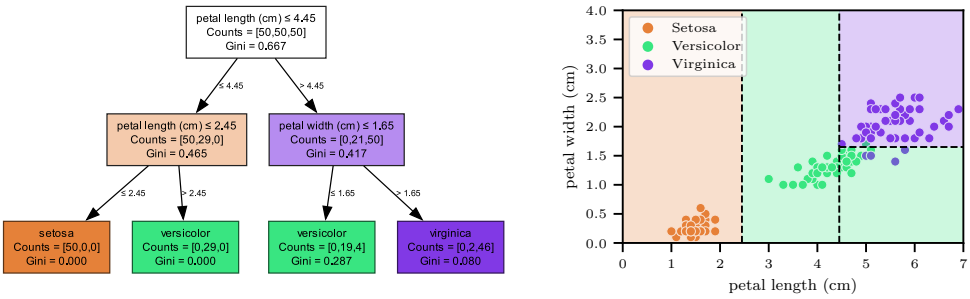
In recent years, machine learning (ML) has become ubiquitous and to such an extent that it is also increasingly applied in high-stakes domains, such as housing appointments (Azizi et al., 2018), criminal justice (Berk et al., 2021), and medical applications (Kononenko, 2001). More than other use-cases, these domains require ML models to be not only *accurate* but also *interpretable*.

Therefore, this thesis studies *optimal decision trees* as a machine learning model that can provide *both* accuracy *and* interpretability. The following introduction further motivates this research, describes (optimal) decision trees, and introduces the challenges to learning these trees. Finally, it presents the research questions and contributions of this thesis.

1.1. Motivation

Accuracy *and* interpretability Traditionally, the main and only objective for ML models is maximizing predictive *accuracy*. However, there is an increasing awareness that high-stakes domains that involve and could potentially harm people require *interpretability* as well (Rudin, 2019). For example, a model that predicts whether a tumor is benign or malignant should obviously be accurate, but to gain experts' trust, *interpretability* is also required. Interpretable models enable experts to understand the output, catch errors, improve models, and make informed decisions. Unsurprisingly, legislators have started making interpretable models mandatory for high-stakes domains (Select Committee on Artificial Intelligence of the National Science & Technology Council, 2019; European Commission, 2021).

While it is typically claimed that accuracy and interpretability are necessarily divergent aims (e.g., Linardatos et al., 2020; Assis et al., 2025), there are plenty of examples where this is not the case. Rudin (2019) point out that specifically for tabular data, simple interpretable models, such as decision trees and rule lists, frequently obtain accuracy on-par with complex black-box models, such as neural networks. Indeed, for tabular data with meaningful features, tree-based methods still outperform even deep learning approaches (Grinsztajn et al., 2022).



(a) (b)

Figure 1.1: (a) An optimal depth-two tree for the Iris dataset. Each node shows the class counts and the corresponding Gini impurity. The node colors and saturation indicate the majority class and the node purity. (b) The colored circles show the true labels of the flowers, based on their petal length and width. The dashed lines are the splits of the tree on the left, and the colored rectangles show the predicted labels of that tree.

These results motivate to search for models that achieve both high accuracy and interpretability, and highlight *decision trees* as a prominent candidate to achieve both these aims. Specifically, this thesis focuses on *optimal decision trees* (ODTs): decision trees that globally optimize a given optimization task (such as maximizing accuracy), while satisfying constraints such as size or depth.

Decision trees A decision tree is a hierarchical structure that recursively divides a total population into smaller subgroups that share common attributes. Starting at the root of a tree, each node in the tree corresponds to an attribute test that separates the total population into its left and right branch. This process is recursively repeated until the final leaf node is reached. Based on, for example, majority vote of the historic (training) population in each leaf node, a label is assigned to it.

Fig. 1.1a shows an example decision tree that labels three different species of the iris flower: setosa, versicolor, and virginica. By splitting the total population on its petal width and length, the three classes are distinguished. Fig. 1.1b shows the location of all 150 training samples and how the decision tree divides these samples into four regions. However, as can be seen, not all training samples are perfectly separated. The aim of decision tree learning therefore is to learn a division of the total sample distribution such that the prediction accuracy is maximized (or equivalently, that the number of errors is minimized), given the available features.

The traditional and most-used approach for decision tree learning is the *greedy top-down induction heuristic*, for example CART (Breiman et al., 1984) and C4.5 (Quinlan, 1993b). These heuristics iteratively find the next (locally) best split to add to the tree model by using a local optimization criterion such as *Gini impurity* or *information gain*. To prevent overfitting on the training data, the trees are typically pruned in a post-processing phase based on how the model scores on a validation set.

Due to their simplicity, these methods are still among the most popular machine learning approaches. Under mild assumptions on the data, optimizing a greedy

decision tree is as cheap as sorting the data, while the output is a simple interpretable model that can accurately capture complex non-linear relationships in the data. As a result, these greedy decision tree heuristics form the backbone of many state-of-the-art ensemble learning methods, such as random forests and boosting ensembles.

Model sparsity However, these traditional approaches do not consider *model sparsity* apart from preventing overfitting, even though it is one of the conditions for *human comprehensibility* (Piltaver et al., 2016), and hence an important aspect of interpretability. Indeed, a small tree as shown in Fig. 1.1 can be understood easily, but a model with thousands of nodes is too large to parse by humans.

Therefore, to obtain interpretable models minimizing the size of a tree is a second objective next to accuracy. For example, in Fig. 1.1, the root node split is redundant. A smaller tree with the same accuracy could be obtained by first splitting on 'petal length (cm) ≤ 2.45' and then split in the right branch on 'petal width (cm) ≤ 1.65'.

The greedy heuristic approach provides no guarantee to find sparse models and can in theory return trees that are an arbitrary factor larger than optimal (Garey and Graham, 1974). Empirically, greedy decision trees are observed to be significantly larger than optimal as well (Murthy and Salzberg, 1995).

Optimal decision trees Decision trees that provably optimize a given learning task on the training data (such as maximizing classification accuracy) under a given size limitation are called *optimal decision trees* (ODTs). This size limitation can be a hard constraint on the depth, number of nodes, or minimum sample size in leaf nodes, but also a soft constraint that, for example, penalizes the addition of each extra node.

Because optimal decision trees optimize both accuracy and size, they are an excellent candidate for interpretable machine learning. Indeed, Rudin et al. (2022) call learning sparse logical models such as decision trees the first *grand challenge* for interpretable machine learning. Therefore, this thesis addresses this challenge and explores new algorithms and applications for optimal decision trees.

1.2. Challenges

To address the grand challenge of learning ODTs, Rudin et al. (2022) list three sub-challenges, all three of which are addressed in this thesis: (i) scalability, (ii) efficiently handling numerical features, and (iii) supporting constraints. Additionally, they mention as a separate grand challenge "understanding, exploring, and measuring the Rashomon set of accurate predictive models", i.e., the set of all close-to-optimal models. We include this as the fourth challenge: (iv) efficiently computing the whole Rashomon set of decision trees. In addition to these challenges, we contribute a fifth: (v) improving our understanding of the differences between greedy and optimal learning approaches. We discuss the five challenges below in the order in which this thesis addresses them.

Constraints and objectives One of the challenges Rudin et al. (2022) mention for learning ODTs is handling constraints more gracefully. For example, when a domain expert expects a monotonic relationship between income and credit risk,

this could be enforced as a global constraint on the model (e.g., Pei et al., 2016). Other common constraints are fairness (Aghaei et al., 2019) and robustness (Vos and Verwer, 2022).

Similar to handling constraints, handling other objectives than maximizing classification accuracy is also an open challenge. As of yet, most research on ODTs is limited to classification while a few consider also regression. But there are many other learning objectives, such as clustering, survival analysis, classification metrics other than accuracy, cost-sensitive classification, and learning prescriptive policies.

Many of these objectives and constraints can easily be added by using *model-based* approaches for learning ODTs, such as mixed-integer programming (MIP), continuous optimization (CO), Boolean satisfiability (SAT), and constraint programming (CP), provided the given constraint can be described in the modeling language. For example, a variety of fairness constraints can easily be imposed on a decision tree learning MIP model (Aghaei et al., 2019). However, as further discussed below, the scalability of these approaches is limited.

On the other hand, *search-based* approaches such as *dynamic programming* have much better scalability, but depend on an assumption of *separability*: it is assumed that the optimal solution to the left and right subtree are independent. However, for many *global* constraints, such as fairness and robustness, this assumption generally does not hold. The open challenge therefore is how to extend these search approaches beyond the basic classification task to support a large variety of objectives and (global) constraints.

Optimal compared with greedy A second challenge is to deepen our understanding of the differences between greedy and optimal approaches to decision tree learning. The last in-depth comparison between the two approaches was done a few decades ago (Murthy and Salzberg, 1995) and recent comparisons between the two show conflicting results: for example, some highlight superior accuracy results of optimal approaches (e.g., Bertsimas and Dunn, 2017) whereas others observe them to be worse (e.g., Zharmagambetov et al., 2021; Marton et al., 2024).

Additionally, most previous comparisons between the two approaches were limited in scope due to lack of scalability. For example, in one of the most extensive recent comparisons, Bertsimas and Dunn (2017) limit most of their experiments either to datasets of at most 100 samples or to trees of at most depth two. Even with these restrictions, their optimal approach frequently did not converge to optimality within the time limit. It is therefore not clear how the two approaches compare in more realistic settings with larger datasets and more permissible size limits.

Scalability Lack of scalability prevented Bertsimas and Dunn (2017) to extend their experiments to larger datasets and size limits, and thus unsurprisingly, Rudin et al. (2022) mention improving scalability as one of the major challenges for learning ODTs. Decision tree optimization is an NP-complete problem (Hyafil and Rivest, 1976), so no efficient (polynomial) algorithm for learning ODTs exists, unless P = NP. However, with increasing computation power, learning ODTs for small datasets has become possible and since the work by Bertsimas and Dunn (2017), who proposed to learn ODTs with MIP, interest in ODTs has resurged, with others also proposing

SAT (Narodytska et al., 2018), CO (Blanquero et al., 2021), and CP approaches (Verhaeghe et al., 2020). Most of these model-based approaches, however, lack scalability beyond small datasets.

Recent algorithmic improvements in search-based approaches, specifically *dynamic programming* (DP) and *branch-and-bound* (BnB), have advanced the scalability of learning ODTs significantly (e.g., Aglin et al., 2020a; Lin et al., 2020; Demirović et al., 2022; Mazumder et al., 2022). These methods exploit the separability of the problem by finding optimal solutions to subtrees independently from one another. Furthermore, lower and upper bounds help to prune significant portions of the search space. Consequently, Costa and Pedreira (2023) mention DP as the most promising approach for ODTs in terms of scalability. Therefore, this thesis follows these approaches as the main direction for further improvement.

However, at the moment, the majority of these search-based ODT methods rely on a *coarse binarization* of the original numerical features to remain scalable, which leads to the next challenge.

Numerical data Rudin et al. consider efficiently handling *numerical features* another major challenge for learning ODTs. Since these challenges were formulated, Mazumder et al. (2022) and Shati et al. (2023b) have contributed a new branch-and-bound and a SAT approach that improve runtime and memory requirements by a large factor compared to previous methods. They also show that optimizing directly with the original features without binarization as a preprocessing step significantly improves out-of-sample accuracy. However, these methods too lack scalability beyond small datasets and/or restrictive size limits. Mazumder et al., for example, explicitly recommend against using their method beyond decision trees of depth three.

The reason why optimizing with numerical data directly is so much harder is that it increases the number of possible trees by a large factor. Suppose a dataset has a numerical feature Age with 60 unique values. This means there are $60-1=59$ unique thresholds τ by which we can split the dataset using the predicate "Age $\leq \tau$" (with τ any one of all values between each two adjacent unique values). Instead, with a coarse binarization, the original data is replaced with binary data that captures only a subset of all possible predicates. For example, as a preprocessing step, one could preselect a limited number of predicates on the Age attribute, such as "Age ≤ 18", "Age ≤ 40", and "Age ≤ 65", thus reducing the number of possible splits from 59 to 3, in this example, and provide one binary feature for each of these splits. Since at each branching node of the tree any one of all possible splits can be selected, the number of trees grows exponentially with respect to the number of possible splits. Therefore, maintaining scalability, while searching over all these possible splits is a major open challenge for optimizing ODTs.

Computing the whole Rashomon set Finally, most ML methods optimize only a *single* model, while many different but close-to-optimal models may exist. Breiman (2001) called this the "Rashomon effect", named after the Japanese movie *Rashomon* that showed a variety of (contradicting) perspectives on the same event. The *Rashomon set* is the set of all models whose performance is close to that of the optimal model. As in the original movie, having multiple perspectives rather

than only one has several advantages, such as enabling discovery of simpler models (Semenova et al., 2022; Rudin et al., 2024), improving feature importance analysis (Donnelly et al., 2023), and provide diverse counterfactual explanations (Andersen et al., 2023).

Since computing even a single optimal model is hard, computing the complete set of all close-to-optimal models is even harder. Therefore, one of the challenges Rudin et al. (2022) mention is to develop techniques that allow us to efficiently compute the Rashomon set. Specifically, they suggest dynamic programming and branch-and-bound as a promising approach to improve the efficiency of computing the whole set. Xin et al. (2022) show that indeed dynamic programming provides major scalability improvements, but also that scalability remains challenging, specifically since the Rashomon set can be huge. Therefore, the challenge remains to find not just one, but all top-performing models efficiently.

1.3. Research Questions

We observe that the *scalability* challenge reoccurs across the other challenges, and therefore we address it as part of the other challenges. We turn each of the four other challenges into a separate research question presented hereafter. We also mention for each research question which chapters contribute an answer. The outline of the thesis follows directly on the order of the research questions.

1.3.1. Objectives and Constraints

This first main research question addresses the challenge of adapting dynamic programming ODT methods to other constraints and objectives than classification.

RQ1. Under what conditions can dynamic programming methods for optimal decision trees be generalized to machine learning tasks beyond classification?

This research question is split up into three subquestions:

RQ1a. What are the precise conditions required for applying dynamic programming to learning optimal decision trees?

Above, we described the condition for applying DP informally as “separability” and therefore to answer the first research question, **Chapter 3** first studies as a test case how a global constraint such as group fairness, which at first seems to not satisfy separability, can be turned into a separable objective. It reuses techniques by Demirović and Stuckey (2021) to obtain a Pareto front of nondominated solutions that are measured both in accuracy and fairness. The result is an algorithm that is orders of magnitude faster than the model-based MIP approach.

Chapter 4 follows up by providing the precise necessary and sufficient conditions for separability, so that DP can be used to optimize ODTs. These conditions are then used to build a general framework to optimize ODTs for a variety of learning tasks, such as cost-sensitive classification, prescriptive policy generation, group fairness, and nonlinear classification metrics.

RQ1b. Which algorithmic improvements to search-based methods (e.g., bounds, efficient preprocessing, etc.) generalize to what other learning tasks and under which conditions?

While separability is the main condition to obtain the scalability performance of DP, other recent algorithmic improvements in search-based methods rely on other assumptions. For example, branch-and-bound typically assumes solution that values to subproblems are additive. Therefore, to maximize scalability, we also need to research what the conditions are to apply these techniques to other learning tasks.

To answer this question, **Chapter 4** also provides conditions on the use of techniques such as subproblem upper bounds, similarity-based bounding, and the depth-two subroutine by Demirović et al. (2022). **Chapters 5 and 6** show how these techniques can be adapted for surival analysis and regression.

RQ1c. How do optimal decision trees perform for other optimization tasks, such as regression and survival analysis, in terms of out-of-sample performance, sparsity, and scalability?

Once we have scalable search algorithms for optimal decision trees for other learning tasks, the remaining question is how these trees perform for each of these tasks in several dimensions such as out-of-sample performance, model sparsity, and scalability. Each of the **Chapters 3-6** contribute to answering this question for a variety of learning tasks with extensive comparisons to greedy and other optimal baselines.

Chapters 3-6 are the results of the following four peer-reviewed papers:

- J.G.M. van der Linden, M.M. de Weerdt, and E. Demirović, "Fair and Optimal Decision Trees: A Dynamic Programming Approach," in *Advances in NeurIPS*, 38899-38911 (2022).
- J.G.M. van der Linden, M.M. de Weerdt, and E. Demirović, "Necessary and Sufficient Conditions for Optimal Decision Trees Using Dynamic Programming," in *Advances in NeurIPS*, 9173-9212 (2023).
- T. Huisman, J.G.M. van der Linden, and E. Demirović, "Optimal Survival Trees: A Dynamic Programming Approach," in *Proceedings of AAAI*, 12680-12688 (2024).
- M. van den Bos, J.G.M. van der Linden, and E. Demirović, "Piecewise Constant and Linear Regression Trees: An Optimal Dynamic Programming Approach," in *Proceedings of ICML*, 48994-49007 (2024).

1.3.2. Comparing Optimal and Greedy Decision Trees

The second research question further studies the differences, strengths, and weaknesses of optimal and greedy decision learning approaches.

RQ2. What are the differences, strengths, and weaknesses of optimal and greedy

decision tree learning approaches?

Again, we split up this question into three smaller subquestions.

RQ2a. What is the effect of the optimization objective on the out-of-sample accuracy and model sparsity for optimal and greedy decision tree learning approaches?

One of the differences between greedy and optimal methods is the objective (or loss function) that is optimized. For top-down greedy induction algorithms, this is typically either Gini impurity or entropy, whereas optimal methods optimize accuracy directly. **Chapter 7** shows that both methods respond differently to varying objectives. Insights from greedy learning can therefore not directly be applied to optimal methods.

RQ2b. What is the effect of the hyperparameter tuning method on the out-of-sample accuracy and model sparsity for optimal and greedy decision tree learning approaches?

A second difference between optimal and top-down greedy approaches is the way model sparsity is decided through hyperparameter tuning. **Chapter 7** compares several hyperparameter tuning methods, both for optimal and greedy approaches, and concludes that the choice of method typically has a small influence on the out-of-sample accuracy, but differences in model sparsity and runtime can be larger.

RQ2c. How do optimal and greedy decision trees compare in terms of accuracy, interpretability, and the trade-off between those, overfitting to noise, adapting to increasing ground-truth model complexity, and how does performance change with more data?

This third sub-question investigates the performance of optimal and greedy approaches across multiple dimensions and in different scenarios. In **Chapter 7**, we collect six claims from the literature on how optimal and greedy approaches compare, and extensively test them on both synthetic and real data. Our results confirm three and refute three claims. Most importantly, we confirm (i) that ODTs on average obtain higher accuracy than greedy trees under a fixed size limit and (ii) that ODTs obtain a better accuracy-interpretability trade-off, while we find no evidence (iii) that differences between ODTs and greedy trees diminish with more data, or (iv) that ODTs would be more prone to overfitting than greedy trees.

All of **Chapter 7** is part of our preprint paper:

➤ J.G.M. van der Linden, D. Vos, M.M. de Weerdt, S. Verwer and E. Demirović, "Optimal or Greedy Decision Trees? Revisiting their Objectives, Tuning, and Performance," *arXiv preprint arXiv:2409.12788* (2024).

1.3.3. Scalability with Numerical Features

The third research question addresses the challenge of scalability when directly optimizing with numerical features. It asks what algorithmic techniques can exploit the nature of numerical data to improve scalability.

RQ3. What algorithmic techniques can be exploited to improve the scalability of learning optimal decision trees directly on numerical feature data?

In answer to this question, we develop in **Chapter 8** several novel techniques. First, we exploit subproblem similarity. E.g., if we have an optimal solution for the data filtered by the predicate "Age ≤ 32", this can probably help solve the next subproblem for the data filtered by the predicate "Age ≤ 33" since the data will differ only slightly. We present three pruning techniques that exploit this property. Second, we exploit the fact that numerical data can be sorted by computing aspects of the problem incrementally. Together these algorithmic techniques improve scalability by up to two orders of magnitude over the state of the art.

Chapter 8 has been published in the following peer-reviewed paper:

➤ C.E. Brita, J.G.M. van der Linden, and E. Demirović, "Optimal Classification Trees for Continuous Feature Data Using Dynamic Programming with Branch-and-Bound," in *Proceedings of AAAI*, 11131-11139 (2025).

1.3.4. Enumerating the Whole Rashomon Set

In the fourth research question, we move beyond a single model and seek to enumerate the whole Rashomon set of close-to-optimal models.

RQ4. What algorithmic techniques can be used to improve the scalability of enumerating the Rashomon set of decision trees in order?

This question explores two challenges: scalability and anytime enumeration of the Rashomon set. Scalability is important to be able to apply the methods to larger datasets. Anytime enumeration is important because the Rashomon set can be huge, and anytime enumeration allows to stop the search at any time and return the Rashomon set computed up to that point in time. By enumerating in order, stopping the search at any time, yields all models closest to the optimal model.

We answer the fourth research question in **Chapter 9**, where we extend the method presented in **Chapter 4** to return all models in the Rashomon set in order. Similar to Xin et al. (2022), we exploit dynamic programming to store optimal solutions to subproblems, which frequently re-appear as partial solutions of many models in the Rashomon set. We also adapt the depth-two subroutine by Demirović et al. (2022) to return all possibly useful partial solutions. Together, these improvements result in up to two orders of magnitudes runtime improvement over the state of the art, while returning the set in-order.

Chapter 9 has been published in the following peer-reviewed paper:

➤ E. Arslan, J.G.M. van der Linden, S. Hoogendoorn, M. Rinaldi, and E. Demirović, "SORTeD Rashomon Sets of Sparse Decision Trees: Anytime Enumeration," in *Advances in NeurIPS* (2025).

1.3.5. Comparing Model- and Search-Based Approaches

Finally, this thesis frequently touches upon the differences between model- (MIP, CO, CP, SAT) and search-based approaches (DP, BnB) to learn ODTs. While we

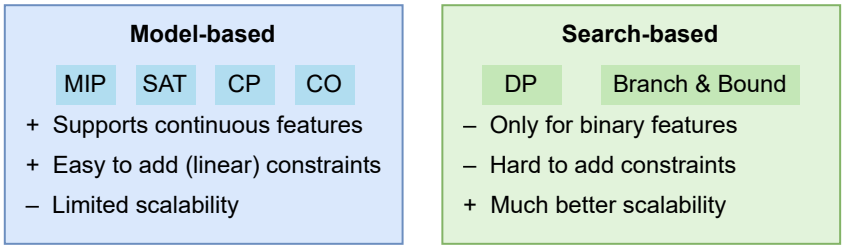


Figure 1.2: The two major solving paradigms for optimal decision trees can be summarized as model-based (MIP, SAT, CP) and search-based (DP and branch-and-bound). The former supports continuous features and adding new constraints or objectives is (typically) easy. However, limited scalability is prohibitive for practical application. Almost all search-based methods, on the other hand, rely on a coarse binarization to remain scalable, but this harms accuracy performance. Additionally, global constraints and alternative objectives are typically hard to enforce. This thesis addresses both weaknesses of search-based approaches.

do not add it as a separate research question, we mention it here as a reoccurring background question.

The discussion of the challenges above reveals how divergent the two approaches are. For example, it reveals that adapting ODT methods to incorporate a variety of (global) objectives and constraints is much easier for model-based than search-based approaches. It also reveals that many of the model-based methods support optimizing on the continuous features directly, whereas most search-based approaches do not. However, the scalability of model-based approaches is severely limited, while this is something the search-based methods do relatively well.

The two solving paradigms for ODTs therefore both have opposite weaknesses, as summarized in Fig. 1.2. It seems though that the advantages of model-based approaches are mostly theoretical, since for real-world application, *scalability* is a non-negotiable requirement and reoccurring in each of the challenges we listed above. The search-based approaches, on the other hand, can scale better, provided a coarse binarization is assumed, and the constraint or objective is separable.

Therefore, this thesis addresses both weaknesses of the search-based approaches: adapting to a large variety of objectives and constraints and scalability improvements for handling numerical features in search.

1.4. Summary of Contributions

By addressing the aforementioned five challenges, this thesis significantly advances our understanding of and methods for training optimal decision trees across several dimensions.

We prove the extent and the limits of the applicability of DP for learning ODTs. We also reveal the breadth of its applicability practically by providing a generalizable framework. This thesis applies the framework to learn ODTs for (i) group fairness constraints, (ii) cost-sensitive classification, (iii), prescriptive policy generation, (iv)

nonlinear classification metrics, (v) survival analysis, and (vi) regression. For these applications we show how significant scalability improvements can be obtained by use of algorithmic techniques, and provide extensive empirical comparison with both greedy and optimal baselines (Chapters 3-6).

We provide an extensive empirical comparison of optimal and greedy methods and show that optimal and greedy learning methods require different types of objectives to preform well. We confirm that ODTs obtain a better accuracy-interpretability trade-off than greedy trees and refute that the differences become smaller with more data and that ODTs are more sensitive to overfitting (Chapter 7).

We improve scalability for optimizing ODTs on numerical feature data by up to two orders of magnitude over the state of the art by applying three new pruning techniques and exploiting the fact that numerical features can be sorted to support incremental computation (Chapter 8).

Finally, we also improve the scalability of computing the Rashomon set of close-to-optimal decision trees by up to two orders of magnitude over the state of the art, while doing so in order of the objective value (Chapter 9).

Together, these advances improve the scalability, applicability, and understanding of optimal decision trees, and contribute to the adoption of inherently interpretable machine learning models to replace black-box alternatives.

2

Background

This section covers the background on optimal decision tree learning and provides the context of the research presented in the consecutive chapters. For recent surveys on decision tree learning, we refer to Ignatiev et al. (2021), Costa and Pedreira (2023) and Blockeel et al. (2023).

Since every chapter refers to and compares with greedy decision tree learning methods, Section 2.1 first provides a brief background on greedy top down induction learning. Section 2.2 then provides more information on the complexity of decision tree learning and includes the proof for the repeated claim in this thesis that learning optimal decision trees is NP-complete. In the next two sections, we survey the search- and model-based approaches to optimal decision trees. Section 2.3 discusses, summarizes, and contrasts the techniques used in state-of-the-art dynamic programming and branch-and-bound approaches. Section 2.4 introduces a variety of model-based approaches. In both sections, we also provide brief benchmarking results of some of the methods, in order to give not only a qualitative but also a quantitative comparison of the methods discussed. Section 2.5 concludes with an overview of applications of optimal decision trees.

The chapters of this thesis are all stand-alone readable, also without this background section. For a more in-depth understanding of the methods proposed in Chapters 3-6 and Chapter 8, we recommend reading Section 2.3 first. We recommend reading the other background sections for a deeper understanding of the research context.

2.1. Greedy Learning

The de-facto default approach to learning decision trees is still greedy top-down induction, as we explain below. Also in this thesis, in every chapter, we compare our new optimal methods with the greedy learning approach. In Chapter 7 specifically, we provide an extensive comparison between optimal and greedy learning approaches. Therefore, we provide here some brief background information on greedy learning.

Early methods Morgan and Sonquist (1963) proposed the first decision tree learning method: Automatic Interaction Detection (AID), a recursive regression analysis approach. Their main motivations for this recursive approach for multivariate analysis were to handle (i) many different variables (features), (ii) categorical variables, (iii) correlations between variables, and (iv) nonlinear (non-additive) relationships in the data.

AID introduced the basic component of the greedy *top-down induction* (TDI) approach. Recursively, based on the available variables, consider all possible ways the data could be split into two groups and compute the reduction in the variance. Find the split with the largest reduction in the variance. If this reduction is smaller than 1% of the variance of the total sample, stop. Otherwise, split the data, and recursively apply the same procedure on both splits.

Kass (1980) adapted AID for classification in Chi-Square Automatic Interaction Detection (CHAID), by replacing measuring the reduction in mean squared error with a χ^2-test, and choosing the split with the highest test statistic. After CHAID, several similar TDI approaches were developed. The most-well known are CART (Breiman et al., 1984), ID3 (Quinlan, 1986), and its follow-up C4.5 (Quinlan, 1993b).

Splitting criterion One of the differences between these approaches is the splitting criterion. CHAID uses the χ^2-statistic, ID3 and C4.5 use information gain (also called entropy), and CART uses the Gini impurity. Given a set of classes $\mathcal{K}$, we have the probability p_k of class $k \in \mathcal{K}$ equal to its relative frequency in a list of labels y:

$$p_k(y) = \frac{1}{|y|} \sum_i \mathbb{1}(y^{(i)} = k) \,.$$

Then, for example, the Gini impurity is:

$$I_{\text{Gini}}(y) = \sum_{k \in \mathcal{K}} p_k(y)(1 - p_k(y)) = \sum_{k \in \mathcal{K}} p_k(y) - p_k(y)^2 = 1 - \sum_{k \in \mathcal{K}} p_k(y)^2 \,.$$

Similarly, the entropy criterion is:

$$I_{\text{Entropy}}(y) = -\sum_{k \in \mathcal{K}} p_k(y) \log(p_k(y)) \,.$$

When splitting the list of labels y into two lists, one of which goes left y_L, and one which goes right y_R, the weighted impurity/information metric is:

$$f(y_L, y_R) = \frac{|y_L|}{|y_L| + |y_R|} I(y_L) + \frac{|y_R|}{|y_L| + |y_R|} I(y_R) \,.$$

Top-down induction algorithm Alg. 1 shows how this splitting criterion is recursively used in CART (and similar algorithms) to find (locally) optimal splits. CART's base case is one of several stopping criteria, e.g., a pure node (all labels are the same), a minimum dataset size is reached, a maximum depth, etc. In all other cases, CART searches over all features $j \in \mathcal{F}$, finds the list of possible thresholds V_j, and for each possible test $x_j \leq \tau$ with $\tau \in V_j$, computes the resulting criterion score using the function f. The dataset is then split according to the (locally) optimal split, while running itself recursively for the two resulting datasets.

Algorithm 1: CART($\mathcal{D}$) recursively finds the best split for a dataset $\mathcal{D} = \left\{\left(x^{(i)}, y^{(i)}\right)\right\}_{i=1}^{n}$ using a splitting criterion f until a stopping criterion is met. Each $x^{(i)}$ is a feature vector with values $x_j^{(i)}$ for each feature $j \in \mathcal{F}$.

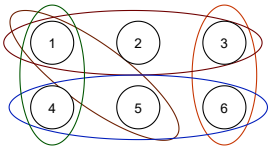
if *stopping criterion (pure node, maximum depth, etc.)* **then**
  return Leaf($\mathcal{D}$)
$s^* \leftarrow \infty$, $j^* \leftarrow 0$, $\tau^* \leftarrow 0$
for $j \in \mathcal{F}$ **do**
  $V_j \leftarrow \left\{\frac{1}{2}\left(x_j^{(i)} + x_j^{(i+1)}\right) \mid i = 1, \ldots, n-1, \text{ sorted by } x_j\right\}$
  for $\tau \in V_j$ **do**
    $y_L \leftarrow \left\{y^{(i)} \mid x_j^{(i)} \leq \tau\right\}$, $y_R \leftarrow \left\{y^{(i)} \mid x_j^{(i)} > \tau\right\}$
    if $f(y_L, y_R) < s^*$ **then**
      $s^* \leftarrow f(y_L, y_R)$, $j^* \leftarrow j$, $\tau^* \leftarrow \tau$
$\mathcal{D}_L \leftarrow \left\{\left(x^{(i)}, y^{(i)}\right) \mid x_{j^*}^{(i)} \leq \tau^*\right\}$, $\mathcal{D}_R \leftarrow \left\{\left(x^{(i)}, y^{(i)}\right) \mid x_{j^*}^{(i)} > \tau^*\right\}$
return Node(j^*, τ^*, CART($\mathcal{D}_L$), CART($\mathcal{D}_R$))

Pruning Typically, CART continues to split nodes until all its leaf nodes are pure (consisting of just one class) or no further splits can be made. When noise is present, such a tree may overfit on the training data. Therefore, the tree is pruned afterwards using *cost-complexity* pruning. For each branching node in the tree, compute the reduction in the misclassification rate s and the number of pruned nodes m, if it were replaced with a leaf node. Then find the node with the lowest value $\alpha = \frac{s}{m}$. The subtree rooted in this node is the subtree with the worst trade-off between accuracy and size. This subtree is replaced with a leaf node, and the procedure continues with finding the next worst subtree, until finally the tree is reduced to a single node. The list of unique α values encountered during pruning is called the *cost-complexity path*. Cross-validation is then used to pick the right complexity cost α.

2.2. Complexity

The advantage of the greedy learning approach is its simplicity and scalability, as we also analyze below. Learning optimal decision trees, on the other hand, is an NP-complete problem and scalability is therefore challenging. In this section, we repeat the NP-completeness proof of ODT learning and discuss other complexity results to further investigate why optimal decision tree learning is hard.

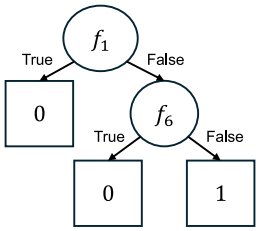
Greedy The greedy approach explained above, if implemented well by using running sums to compute the splitting criterion value, has a runtime complexity of $\mathcal{O}(|\mathcal{F}||\mathcal{D}| \log |\mathcal{D}|)$, under the assumption that the procedure produces (close to) balanced splits. This complexity is obtained by sorting the dataset once for each feature at the beginning. Balanced splits result in a tree of depth $\mathcal{O}(\log |\mathcal{D}|)$ and since each sample appears in precisely one node in every layer of the tree, the complexity of finding the best splits in each layer is $\mathcal{O}(|\mathcal{F}||\mathcal{D}|)$, thus also yielding $\mathcal{O}(|\mathcal{F}||\mathcal{D}| \log |\mathcal{D}|)$.



(a)

	f_1	f_2	f_3	f_4	f_5	f_6	y
$x^{(1)}$	1	1	1	0	0	0	0
$x^{(2)}$	1	0	0	0	1	0	0
$x^{(3)}$	1	0	0	1	0	0	0
$x^{(4)}$	0	0	0	1	1	1	0
$x^{(5)}$	0	0	1	0	0	1	0
$x^{(6)}$	0	0	0	0	0	0	1

(b)



(c)

Figure 2.1: (a) An example of a hitting set problem (b) turned into a decision table, and (c) a tree of minimum size that perfectly captures the decision table. This shows that selecting object 1 and 6 in (a) is a hitting set of minimum size.

In the worst case, each split only splits off a single sample, resulting in a tree of depth $|\mathcal{D}|$. Therefore, the worst-case complexity of Algorithm 1 is $\mathcal{O}(|\mathcal{F}||\mathcal{D}|^2)$.

NP-complete However, this algorithm does not guarantee an *optimal* solution, i.e., a tree of *minimum size* that correctly classifies all training samples. Finding such an optimal tree is shown to be an NP-complete problem. Hyafil and Rivest (1976) showed this by a reduction from the three-set exact set cover. Later, Cox et al. (1989) proved that finding an optimal tree (that minimizes feature inspection costs) that perfectly represents a linear function, is also NP-complete by reducing the knapsack problem to the subproblem of finding the optimal root node split. Similarly, Murphy and McCraw (1991) showed that finding a decision tree of minimum size is NP-complete by a reduction from vertex cover, even if one of the classes is represented by only one sample.

Reduction from hitting set Ordyniak and Szeider (2021) prove that optimizing decision trees is NP-complete by a reduction from the hitting set problem. Let $U = \{u_1, u_2, ..., u_n\}$ be a finite universe of objects and $\mathcal{S} = \{S_1, S_2, ..., S_m\}$ a finite set of sets consisting of elements of U, i.e., $S_j \subseteq U, \forall j \in [m]$. The hitting set problem asks if there is a subset $V \subseteq U$ of at most size k such that V contains at least one element from each set in $\mathcal{S}$, i.e., $\forall j \in [m] : |V \cap S_j| > 0$. See Fig. 2.1a for an example hitting set instance.

This problem can be reduced as follows to the decision problem whether an optimal decision tree of at most k branching node exists. For each set S_j, construct a sample $(x^{(j)}, y^{(j)})$ with as feature vector $x_i^{(j)} = 1$ if $u_i \in S_j$ and 0 otherwise, and let $y^{(j)} = 0$. Finally, create one sample $(x^{(m+1)}, y^{(m+1)})$ with $x^{(m+1)}$ the all-zero vector, and $y^{(m+1)} = 1$. See Table 2.1b for how the example hitting set instance can be transformed into a decision table. Each row is a sample and represents a set S_j from Fig. 2.1a (except the last row, which is the special case). Each object u_i is transformed into a feature column f_i, with the additional row for the special case.

Since there is only one positive sample ($y = 1$), an optimal decision tree will only have splits that have a zero-label leaf node on one side (since each split produces at least one pure leaf). Therefore, the decision problem is how many feature tests are

needed to distinguish the one positive sample from each negative sample ($y = 0$). This is the same as solving the hitting set problem, because each split on a feature f_i splits of a (number of) row(s) with label zero, which is the same as hitting the corresponding set S_j by selecting object u_i. Therefore, if it can be decided in polynomial time that a decision tree of size k exists, then the hitting set problem can also be decided in polynomial time. Since the hitting set is an NP-complete problem, so is the optimal decision tree problem. See Fig. 2.1c for the optimal decision tree for the decision table in Table 2.1b. Selecting u_1 and u_6 is the optimal hitting set for Fig. 2.1a and splitting on f_1 and f_6 is the optimal tree for the decision Table 2.1b.

Further complexity results From this reduction, Ordyniak and Szeider (2021) also conclude other complexity results. As the reduction above shows, the same reduction also works to show that finding trees of minimum depth is NP-complete. Furthermore, they reason when the problem becomes fixed-parameter tractable (FPT). The considered parameters are tree size (either depth or number of nodes), the maximum domain size (the number of unique values for a feature), and the maximum distance (the maximum hamming distance between any two samples with different classes). Only by fixing all three of these parameters, the problem becomes tractable. Thus, the problem also remains intractable for boolean features and for fixed maximum depth (both of which are recurring assumptions in this thesis).

2.3. Search-Based Solving

All ODT methods proposed in this thesis are *search-based* using a combination of dynamic programming and branch-and-bound techniques. In Chapters 3 and 4, we show how existing dynamic programming approaches can be generalized to many more learning tasks, and Chapters 5 and 6 give two more concrete examples of this. In Chapter 8 we present a dynamic programming and branch-and-bound algorithm for ODT learning with numerical feature data. All these chapters build on previous research in search-based ODT learning and therefore we provide in-depth background information on these previous methods. To motivate our choices, we also provide a short experimental comparison between existing search-based approaches.

2.3.1. Dynamic Programming

Among the first decision tree learning methods, we immediately find optimal dynamic programming (DP) approaches (Schumacher and Sevcik, 1976; Payne and Meisel, 1977; Miyakawa, 1985; Cox et al., 1989). DP naturally captures the inherent *recursion* and *separability* in the problem: any optimal decision tree (beyond a single leaf node) consists of an optimal root split and two optimal subtrees. In recent years, this natural problem decomposition has been combined with branch-and-bound pruning techniques.

Consider a classification problem for a dataset of n samples $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}_{i=1}^{n}$ with $x^{(i)} \in \{0,1\}^{|\mathcal{F}|}$ a binary feature vector for each binary feature in $\mathcal{F}$, and $y^{(i)} \in \mathcal{K}$ the label for that sample. Then, following the notation from Demirović et al. (2022), let $\mathcal{D}(f) = \{(x, y) \in \mathcal{D} \mid x_f = 1\}$ capture the samples in $\mathcal{D}$ that satisfy feature f

and similarly $\mathcal{D}(\bar{f}) = \{(x, y) \in \mathcal{D} \mid x_f = 0\}$ those that do not. Then the minimum number of misclassifications for a tree of maximum depth d for $\mathcal{D}$ can be found with the recursion:

$$\text{OPT}(\mathcal{D}, d) = \begin{cases} \min_{\hat{y} \in \mathcal{K}} \sum_{(x,y) \in \mathcal{D}} \mathbb{1}(\hat{y} \neq y) & \text{if } d = 0 \\ \min_{f \in \mathcal{F}} \left(\text{OPT}(\mathcal{D}(f), d-1) + \text{OPT}(\mathcal{D}(\bar{f}), d-1) \right) & \text{else.} \end{cases} \tag{2.1}$$

This recursion captures the basic DP approach. It can easily be extended to also optimize over the total number of nodes or by adding a penalty λ for each node that is added (the *sparse* objective).[1]

The recent literature shows a number of improvements on the basic recursion that, on average, improve runtime performance. These are (i) caching, (ii) bound-based pruning, (iii) alternative search strategies, (iv) special subprocedures for small subproblems, (v) anytime performance, and (vi) handling of continuous features.

2.3.2. Caching

Caching is an essential part of any dynamic programming approach. In Eq. (2.1), a subproblem is identified by the dataset $\mathcal{D}$ and the size budget d. To check the equivalence of two such subproblems, different techniques have been used. We describe each below.

Branch Any dataset $\mathcal{D}$ encountered in the subproblems is the result of filtering the original dataset by specific feature values, namely those feature values tested in the parent branching nodes of the currently evaluated node. For example, if the path to the current node filtered on $f_5 = 0$ and $f_3 = 1$, we could represent this as the *branch* $\{(f_5, 0), (f_3, 1)\}$. This set is order-invariant since it does not matter in which order we filter the dataset. Therefore, a subproblem is entirely identified by just the numbers of the features and their values branched on, which is cheap to compute and store in memory. Cache storage is done either using a *trie* or a *hash map*.

However, branch caching does not identify all equivalent subproblems. For example, branching on either of two equivalent features results in identical subproblems, but not in identical branches.

Closure To improve the recognition of equivalent subproblem, Nijssen and Fromont (2007, 2010) propose to cache subproblems by their *closure*: the largest set of features that all instances in a dataset either all satisfy or do not satisfy. Formally:

$$\text{Closure}(\mathcal{D}) = \cap_{(x,y) \in \mathcal{D}} \{(f, v) \in \mathcal{F} \times \{0, 1\} \mid x_f = v\} \ .$$

The closure of a dataset is always a superset of the branch set. Obviously, if $\mathcal{D}_1 = \mathcal{D}_2$, then also $\text{Closure}(\mathcal{D}_1) = \text{Closure}(\mathcal{D}_2)$.

However, Demirović et al. (2022) observe that computing the closure incurs a significant overhead and advice against using it.

[1] Lin et al. (2020) penalize *leaf* nodes, whereas we penalize *branching* nodes. For binary trees (the only trees considered in this thesis) the difference is only a constant λ. Also, Lin et al. consider *normalized* losses, whereas we consider the misclassification score directly. Therefore, to get the same regularization effect, the λ parameter must be multiplied by the dataset size $|\mathcal{D}|$.

Bitset Lin et al. (2020) identify a dataset by the identifiers i of the samples in it. This is stored in a bitset such that the ith bit is one if and only if the ith sample is part of the currently considered dataset.

Bitset caching identifies all equivalent subproblems. Computing the bitset is more expansive than computing the branch representation of the dataset but easier than computing its closure. While the bitset representation is a cheap encoding of the data (it requires only n bits), the cost is constant. For example, the encoding of a subset of the data of only one sample still requires the same amount of data as the whole dataset.

List Demirović et al. (2022) also identify a dataset by the identifiers of its samples, but store these identifiers in a list rather than in a bitset. However, since they provide a special base case for subproblems of at most depth two (see Section 2.3.5), their total number of cache entries is far fewer than other approaches and memory usage by the cache is less of a concern.

Demirović et al. (2022) observe better runtime performance for their list (dataset) caching than branch caching. However, these results are based on datasets with several equivalent binary features as a result of the binarization approach used. For example, the binarization resulted in both the feature $f_1 = [\text{Age} \leq 18]$ and $f_2 = [\text{Age} > 18]$, such that $\mathcal{D}(f_1) = \mathcal{D}(\bar{f}_2)$ (note the negation). If such equivalent features are removed in preprocessing, the number of equivalent subproblems for different branches may drop, and therefore, the performance trade-off between branch and dataset caching after removing such redundant features remains an open question.

Memory constraints Aglin et al. (2022) explore training optimal decision trees under memory constraints by setting a maximum cache size. Whenever the cache is full, a percentage of the cache is erased. Their experiments show that retaining cache entries for the current optimal solution and the most used cache entries helps to keep runtime close to the performance when no cache entries are erased.

The methods proposed in this thesis use both *branch* caching and *dataset* caching using bitsets. Since we use similar depth-two subprocedures as Demirović et al. (2022), we also seldom encounter lack of memory and therefore do not further explore methods to reduce the cache memory footprint.

2.3.3. Bound-Based Pruning

To reduce the search space and improve runtime performance, Aglin et al. (2020a) incorporated branch-and-bound search in the recursive function. Each recursive call can be considered a search node in a branch-and-bound search. By computing and storing lower and upper bounds (LB and UB respectively) for each such search node, a search node can be pruned if $LB \geq UB$. Each solution (of a subproblem) is an upper bound on the optimal solution. Good lower bounds, however, are harder to obtain. Here, we discuss a number of bounding techniques proposed in the literature.

Hierarchical Aglin et al. (2020a) show that if you have a current UB (e.g., an incumbent solution), and an optimal solution for one side of a split, e.g.,

$\text{OPT}(\mathcal{D}(f), d-1)$, an upper bound for the other split can be computed by subtracting this value from the incumbent:

$$UB(\mathcal{D}(\bar{f}), d-1) = UB(\mathcal{D}, d) - \text{OPT}(\mathcal{D}(f), d-1)\,. \tag{2.2}$$

However, this changes the definition of an upper bound: no longer is the upper bound the best value we have found so far (the incumbent) but the upper bound now indicates the solution value we need to obtain to improve the current incumbent (possibly of the parent node). Therefore, we can prune a node only if $LB > UB$ (rather than $LB \geq UB$). This also means that (for classification) the upper bound can be set to the incumbent solution minus one.

Demirović et al. (2022) further improve on this reasoning in two ways: (i) Eq. (2.2) can also be used if no optimal solution for one side is found yet. Instead of subtracting the optimal solution, one can also subtract the lower bound; (ii) if no solution can be found below the given upper bound, this upper bound can be stored in the cache as a lower bound.

Lin et al. (2020) use similar reasoning as Aglin et al. (2020a), but call this approach the hierarchical *lower* bound, since they consider the cost of a subtree as a lower bound for the cost of the whole tree.

Similarity Hu et al. (2019) introduce the *similar support* bound (in this thesis referred to as the *similarity* lower bound) by observing that if two (sub)trees are precisely the same but evaluated on two different datasets $\mathcal{D}_1$ and $\mathcal{D}_2$, then the size of the difference $\mathcal{D}' = (\mathcal{D}_1 \cup \mathcal{D}_2) \setminus (\mathcal{D}_1 \cap \mathcal{D}_2)$, can be used to formulate a lower bound. Demirović et al. (2022) strengthen this bound by not considering all differences in $\mathcal{D}'$, but only those samples that are present in $\mathcal{D}_1$, but not in $\mathcal{D}_2$. This assumes the worst-case scenario that all samples removed from $\mathcal{D}_1$ were misclassified by the optimal tree for $\mathcal{D}_1$, while also assuming that all samples that are added will be classified correctly by that same tree. This yields the bound:

$$\text{OPT}(\mathcal{D}_2, d) \geq \text{OPT}(\mathcal{D}_1, d) - |\mathcal{D}_1 \setminus \mathcal{D}_2|\,. \tag{2.3}$$

Lin et al. (2020) exploit this bound for splitting on binary features that represent different splitting thresholds for continuous features. They also show how the bound can be used for other objectives that are a linear function of the false positives and negatives. Demirović et al. (2022) compute the difference of the current subproblem dataset to the last two previously handled subproblems. For both methods, the cost of computing the difference is $\mathcal{O}(|\mathcal{D}|)$.

Mazumder et al. (2022) use similar reasoning to bound the error of a dataset, when we have a solution for a dataset where a *subinterval* of the samples is missing (the data here is sorted by some continuous feature). Consider $\mathcal{D}_L = \mathcal{D}(f \leq a)$ and $\mathcal{D}_R = \mathcal{D}(f > b)$, with $a < b$, such that we have already computed $\text{OPT}(\mathcal{D}_L, d-1)$ and $\text{OPT}(\mathcal{D}_R, d-1)$. If $\text{OPT}(\mathcal{D}_L, d-1) + \text{OPT}(\mathcal{D}_R, d-1) > UB$, and since by the similarity lower bound, the addition of any samples from $\mathcal{D}(a < f \leq b)$ cannot improve the error, then any split with a threshold $a < \tau \leq b$ cannot provide an improving solution.

Equivalent points The *equivalent points* lower bound considers samples (x, y) and (x', y') that have the same feature vector ($x = x'$), but disagree in their label ($y \neq y'$). Since they share the same feature vector, there is not a single split that can distinguish the two samples. Therefore, the two samples are guaranteed to end up in the same leaf node, and at least one of them must be misclassified.

Hu et al. (2019) therefore collect all samples with the same feature vector and increase the lower bound adding the size of the minority class in each set of samples with the same label. If the samples are sorted in a preprocessing step based on their feature values, then checking for equivalent feature vectors can be done in $\mathcal{O}(n)$ in each search node.

Alternatively, as a preprocessing step, Demirović et al. (2023) remove samples that cancel each other out. This reduces the dataset size and improves runtime. However, information on the true number of samples in leaf nodes are lost, and for example, enforcing a minimum leaf node size constraint after the samples are removed cannot easily be done anymore.

Zhang et al. (2023) adapt the equivalent points lower bound for regression by grouping samples with the same feature vector. Such samples are grouped, and the sum of the squared errors with respect to the mean of the sample group is a lower bound on the error for these samples.

Sparsity-based Hu et al. (2019) and Lin et al. (2020) offer additional bounding techniques when the objective also contains a regularization parameter λ for penalizing the addition of nodes. Let $\text{Leaf}(\mathcal{D}) = \text{OPT}(\mathcal{D}, 0)$ be the misclassification cost of a leaf and let $\text{Equiv}(\mathcal{D})$ be the equivalent points lower bound, then combining these, the *one-step lookahead* lower bound for a subproblem with $d > 0$ is:

$$\text{OPT}(\mathcal{D}, d) \geq \min(\text{Leaf}(\mathcal{D}), \lambda + \text{Equiv}(\mathcal{D}))\,. \tag{2.4}$$

This means that the solution to any subproblem is bounded by either the leaf solution, or by requiring at least one branching node plus the equivalent points lower bound obtained from the samples that cannot be separated. Chaouki et al. (2025) present a simplified version of this bound that does not include the equivalent points lower bound.[2]

When minimizing the regularized misclassification error, with regularization penalty λ, there are further ways to prune the search space. Given an upper bound UB, the maximum number of branching nodes is $\lfloor UB/\lambda \rfloor$. Since the maximum depth is also bounded by the number of branching nodes, it also holds that the maximum depth $d = \lfloor UB/\lambda \rfloor$.

Using the regularization penalty λ, we can also provide a lower bound for the size of each leaf node (number of training samples that should reach each leaf node). Because of the similarity lower bound, we need to split off at least $\lceil \lambda \rceil$ samples from the parent node to reduce the misclassification score by that same amount. Since adding the split adds λ to the cost, this is only worthwhile if the minimum leaf node size is $\lceil \lambda \rceil$. Using similar reasoning: any binary feature for which less than $\lceil \lambda \rceil$ of the samples are satisfied, can be ignored.

[2] Chaouki et al. (2025) present their *purification bound* as an improvement on the one-step lookahead bound by Lin et al. (2020), but their bound is in fact less tight.

Algorithm 2: MurTree's core DFS recursion given a dataset $\mathcal{D}$, a depth budget d and an upper bound UB. Some optimization details are left out.

```
MurTree(D, d, UB) :
    // Prune based on the uper bound.
    if UB < 0 then return ∅
    // Use a single leaf node as an initial solution.
    T_best ← Leaf(D)
    if L(T_best, D) > UB then T_best ← ∅
    // If no depth budget left, return the leaf solution.
    if d = 0 then return T_best
    // Check if a cached solution is within the upper bound.
    T_cache ← cache.OPT(D, d)
    if T_cache ≠ ∅ ∧ L(T_cache, D) ≤ UB then return T_cache
    else if T_cache ≠ ∅ then return ∅
    // Obtain the lower bound from cache, or zero if not found.
    LB ← cache.LB(D, d)
    if LB > UB then return ∅
    for f ∈ F do
        // Stop if no improvement can be made.
        if L(T_best, D) = LB then break
        // Ignore splits that produce empty subtrees.
        if |D(f)| = 0 ∨ |D(f̄)| = 0 then continue
        // Set the upper bound to search for improving solutions.
        UB' ← min(UB, L(T_best, D) − 1)
        // Set the subproblem upper bounds and solve them.
        UB_L ← UB' − cache.LB(D(f), d − 1)
        T_L ← MurTree(D(f̄), d − 1, UB_L)
        if T_L = ∅ then continue
        UB_R ← UB' − L(T_L, D(f))
        T_R ← MurTree(D(f), d − 1, UB_R)
        if T_R = ∅ then continue
        // If improving subtrees were found, update the best solution.
        T_best ← Tree(f, T_L, T_R)
    // If a solution was found, store it in the cache.
    if T_best ≠ ∅ then cache.OPT(D, d) ← T_best
    // Otherwise, store the upper bound as the new lower bound.
    else cache.LB(D, d) ← max(LB, UB + 1)
    return T_best
```

2.3.4. Search Strategy

Another difference between state-of-the-art DP methods is the search strategy employed. Here, we cover depth-first, best-first, and AND-OR search.

Depth-first The most intuitive search strategy is to take the recursive formula of Eq. (2.1), and to search it *depth-first*, i.e., to fully expand the current subproblem before considering any other subproblem. This is the approach taken by DL8.5 (Aglin et al., 2020a) and MurTree (Demirović et al., 2022). The advantages of depth-first search (DFS) are (i) a (relatively) low memory footprint; (ii) tight bounds for the second subtree after the first is computed optimally; and (iii) similar subproblems are typically explored sequentially (which can be exploited by techniques such as the similarity lower bound). The disadvantages are (i) that only one branch is expanded initially, which can harm anytime performance; and (i) that the search may overcommit to a subproblem while another subproblem could have yielded good solutions earlier.

Algorithm 2 shows the core procedure of DFS as implemented in MurTree (although several details are left out). The algorithm first checks if the upper bound indicates a feasible subproblem. If so, it constructs the leaf node solution and checks its misclassification score using $\mathcal{L}$. This solution is returned if there is no depth budget left. Otherwise, it continues by checking the cache and returns the cached solution if it is found and within the upper bound. If not, it checks the cache for a lower bound, and checks if the lower bound is not higher than the upper bound.

It then starts the main loop of the recursion: for each feature the following steps are carried out. If the current solution is equal to the lower bound, it quits the search. If the feature split yields an empty left or right dataset, it is skipped. The algorithm then computes the upper bound for the left subproblem by subtracting the lower bound for the right subproblem, and then recursively solves this left subproblem. If no tree is found within the upper bound, it skips to the next feature. Similarly, it solves the right subproblem. If both a left and right subtree have been found, the best solution is updated.

Finally, after going over all features, the optimal tree (if one is found within the upper bound), is stored in the cache. Otherwise, the original upper bound is stored in the cache as a lower bound.

The runtime complexity of this algorithm is $\mathcal{O}(|\mathcal{D}|(2|\mathcal{F}|)^d)$: for each subproblem, we need to split the data $|\mathcal{F}|$ times, each of which costs $\mathcal{O}(|\mathcal{D}|)$, and create $2|\mathcal{F}|$ subproblems. Each of these new subproblems, may also create $2|\mathcal{F}|$ subproblems, iteratively up to the maximum depth d. If we keep track of the label counts whenever we split the data, finding the optimal label in a leaf only takes $\mathcal{O}(|\mathcal{K}|)$, but since typically $|\mathcal{K}| \ll |\mathcal{D}|$, we can ignore this. Therefore, there are $\mathcal{O}((2|\mathcal{F}|)^d)$ subproblems to be explored, each of which costs $\mathcal{O}(|\mathcal{D}|)$ of work, resulting in the complexity $\mathcal{O}(|\mathcal{D}|(2|\mathcal{F}|)^d)$.

For the omitted details, we refer to the original paper by Demirović et al. (2022). They further include a node budget in addition to the depth budget; a special sub-procedure for when the depth budget is at most two (see Section 2.3.5); how and when the similarity lower bound is computed and updated; how a local improved lower bound can be computed by combining the lower bounds for all subproblems; and how to dynamically change the order of the left and right subproblem.

MurTree's DFS algorithm forms the basis for the algorithms proposed in Chapters 3-5.

Algorithm 3: GOSDT's core BFS algorithm iteratively pops subproblems from the queue and expands them, and if the bounds are updated, propagates them back by reinserting its parents into the queue.

```
r ← 𝒟, Q ← {r} // Initialize the queue with the root problem.
// Continue searching until optimality is proven.
while LB(r) < UB(r) do
    s ← Q.pop() // Pop a subproblem from the queue.  Skip if solved.
    if LB(s) = UB(s) then continue
    lb, ub ← ∞, ∞ // Initialize new bounds.
    for f ∈ ℱ do
        s_L, s_R ← Split(s, f) // Split the data on f.
        lb ← min(lb, LB(s_L) + LB(s_R)) // update local bounds.
        ub ← min(ub, UB(s_L) + UB(s_R))
    // Signal the parents if an update occurred (prioritized).
    if LB(s) ≠ lb ∨ UB(s) ≠ ub then
        UB(s) ← min(UB(s), ub)
        LB(s) ← min(UB(s), max(LB(s), lb))
        for p ∈ parents(s) do Q.push(p, prioritized)
    if LB(s) = UB(s) then continue
    // Put children in the queue (if not closed yet).
    for f ∈ ℱ do
        s_L, s_R ← Split(s, f)
        lb ← LB(s_L) + LB(s_R)
        ub ← UB(s_L) + LB(s_R)
        if lb < ub ∧ lb ≤ UB(s) then
            Q.push(s_L)
            Q.push(s_R)
```

Best-first Instead of depth-first search, GOSDT (Lin et al., 2020) uses best-first search (BFS) with a global priority queue that decides which subproblem is expanded next based on a priority heuristic. Lin et al. consider this one of the key differences between GOSDT and DL8.5. The advantages of BFS are (i) more flexible choice of search order; (ii) more balanced exploration of left and right subproblems; and (iii) faster bound propagation to parents and siblings. The disadvantages are (i) that storing all open subproblems in the queue gives a higher memory footprint; and (ii) that the search quality is dependent on the priority heuristic, such as a subproblem's lower bound, which is possibly not sufficiently informative.[3]

Algorithm 3 presents the core of the GOSDT algorithm. It starts out with a single problem in the queue, identified by the whole dataset $\mathcal{D}$. It then continues looping over problems in the queue as long as the gap between the root search node's lower and upper bound is not closed. Whenever Alg. 3 mentions $LB(s)$ or $UB(s)$

[3]Lin et al. (2020) do not mention their heuristic in the paper. The code reveals they use $|\mathcal{D}| - LB$ (search node support minus lower bound), with priority given to higher values. The effect of this heuristic, and which heuristic is best, has not yet been studied.

it refers to the cached values for that subproblem (identified by its dataset). If the subproblem is not encountered in the cache, the lower bound is initialized with for example the similarity lower bound or the sparsity-based bounds, and the upper bound is initialized with the leaf node solution.

Each node in the queue is only processed if it still has an open gap between the lower and upper bound. If so, new lower and upper bounds lb and ub are computed by iterating over all possible feature splits. The new bounds are the minimum of the sum of left and right bounds for all possible splits. If this results in different bounds than previously stored in the cache, the bounds in the cache are updated, and the parents of the current search node are re-inserted into the queue, so that the parents' bounds can also be updated. If the gap is closed after the update, the subproblem is closed, otherwise the algorithm iterates again over all possible splits and inserts the left and right children of the splits where the bounds indicate finding improving solutions is possible. This procedure continues until the gap in the root node is closed and hence a proof of optimality is obtained.

AND-OR The decision tree problem actually does not fit precisely in the (global) best first search paradigm presented above. Best-first search generally assumes traversal over an OR graph, i.e., at each search node in the graph you *either* take this route *or* the other, but not both. Shortest path and A^*-search are a typical example. The decision tree problem, is represented by an AND-OR graph. In OR search nodes, *either one* of the outgoing paths must be chosen (e.g., split on feature f_1 or f_2), whereas in AND search nodes *both* subpaths must be taken (i.e., evaluate both the left and right subproblem. The solution to OR-graph problems is a path through the graph, but for AND-OR graphs, the solution is a tree in the graph. Consequently, a search node (which implicitly describes a path to that node in the graph) in OR search describes a partial path, but in AND-OR search only describes *one branch* out of several branches in the solution tree.

Martelli and Montanari (1978) first applied the combination of DP and AND-OR search to the decision tree problem. They conclude that the effectiveness of the approach depends on the quality of the lower bound, and is in the worst-case less efficient than depth-first DP, but with a proper lower bound, could potentially search fewer nodes. More recently, the AND-OR search paradigm is also applied: Verhaeghe et al. (2020) use it in their constraint-programming approach to ODTs;[4] Sullivan et al. (2024) propose a Bayesian perspective in decision tree optimization using AND-OR search; and Chaouki et al. (2025) use it to formalize the decision tree problem as an Markov Decision Problem (MDP).

The main difference with the best-first search presented above is that each search node has its own priority queue instead of one global priority queue. Selecting the next node to expand starts from the root search node. It selects the node with the lowest lower bound. If expanded, it recursively selects its child node. If not, the node is expanded. Expanding means to create a priority queue in that search node with each possible split in that queue. The new upper and lower bounds are then

[4] In the description used here, the CP approach by Verhaeghe et al. (2020) better fits the description of a depth-first approach.

back-propagated along the path to the root. The new upper and lower bounds are the minimum of the bounds of the children.

This approach ensures that each expansion always expands a node that currently directly contributes to the global lower bound, and Chaouki et al. (2025) report optimistic results for small datasets. However, this approach strongly depends on the quality of the lower bound, and if the search space is large and the lower bounds are poor, easily runs into memory problems.

2.3.5. Depth-Two Case

Demirović et al. (2022) contributed a special subroutine for solving trees of at most depth two that significantly reduces the runtime. It exploits the fact that many feature vectors are sparse. In fact, by flipping any binary feature which is satisfied by more than 50% of the training samples, the average number of satisfied features in any feature vector will always be 50% or less. Then, for each sample, they only loop over the features that are satisfied, to precompute counts.

Formally, considering binary classification, define:

$$FQ^+(f) = |\{(x, y) \in \mathcal{D} \mid x_f = 1 \wedge y = 1\}| \tag{2.5}$$
$$FQ^-(f) = |\{(x, y) \in \mathcal{D} \mid x_f = 1 \wedge y = 0\}| \tag{2.6}$$
$$FQ^+(f, g) = |\{(x, y) \in \mathcal{D} \mid x_f = 1 \wedge x_g = 1 \wedge y = 1\}| \tag{2.7}$$
$$FQ^-(f, g) = |\{(x, y) \in \mathcal{D} \mid x_f = 1 \wedge x_g = 1 \wedge y = 0\}|\,. \tag{2.8}$$

These counts can be computed efficiently by looping over the sparse vector representation for each possible and negative sample. $FQ^+(f, g)$ thus represents the number of positive samples that satisfy both feature f and g. We can now implicitly compute other counts, such as $FQ^+(f, \bar{g})$, the number of positive samples that satisfy feature f, but not g:

$$FQ^+(\bar{f}) = |\mathcal{D}^+| - FQ^+(f) \tag{2.9}$$
$$FQ^+(f, \bar{g}) = FQ^+(f) - FQ^+(f, g) \tag{2.10}$$
$$FQ^+(\bar{f}, g) = FQ^+(g) - FQ^+(f, g) \tag{2.11}$$
$$FQ^+(\bar{f}, \bar{g}) = |\mathcal{D}^+| - FQ^+(f) - FQ^+(g) + FQ^+(f, g)\,. \tag{2.12}$$

Here, $|\mathcal{D}^+|$ denotes the total number of positive samples. Counts of the negative samples can be computed analogously.

With the counts computed, Alg. 4 shows how Demirović et al. compute optimal depth-two trees. For each feature pair, it assigns feature f as the root node split and feature g as the split on either left and right subtree and computes the errors of all four leaf nodes. The error of the left and right split is then computed as the sum of its two children. If this is better than what was previously stored, the best left or right split (given root feature f) is updated. In the end, the algorithm loops once more over all possible root splits and selects the root split with the lowest error.

Chapters 3-6 explore how this subroutine can be applied to other optimization tasks than classification.

Algorithm 4:

MurTreeDepthTwo($\mathcal{D}$) :
  ComputeCounts($\mathcal{D}$)
  for $f \in \mathcal{F}$ **do**
    for $g \in \mathcal{F}$ s.t. $f \neq g$ **do**
      `// Compute the error if` g `is satisfied, but` f `is not.`
      $C(\bar{f}, g) \leftarrow \min(FQ^+(\bar{f}, g), FQ^-(\bar{f}, g))$
      `// Compute the error if neither` f `or` g `is satisfied.`
      $C(\bar{f}, \bar{g}) \leftarrow \min(FQ^+(\bar{f}, \bar{g}), FQ^-(\bar{f}, \bar{g}))$
      `// Compute the left branch error.`
      $C_L \leftarrow C(\bar{f}, g) + C(\bar{f}, \bar{g})$
      `// Check if this error is better than we had before.`
      if $C_L < BestLeftSubtree(f).C$ **then**
        $BestLeftSubtree(f).C \leftarrow C_L$
        $BestLeftSubtree(f).feature \leftarrow g$
      `// Compute the best right tree analogously.`
      ...
  return $\text{argmin}_{f \in \mathcal{F}}(BestLeftSubtree(f).C + BestRightSubtree(f).C)$

2.3.6. Anytime Performance

In addition to finding the optimal solution and proving its optimality as fast as possible, it is also important to obtain good *anytime* performance. Anytime performance means that the algorithm finds good solutions early during the search so that if the search is stopped at any time, a good solution can be returned. The DFS approach above, for example, does not optimize for anytime performance, since it exhaustively explores one subtree first before opening the second branch. As a result, good global solutions may only be obtained after a long time.

Kiossou et al. (2022) circumvent this issue by *limited discrepancy search* (Harvey and Ginsberg, 1995), which starts with the greedy heuristic solution and iteratively increases the discrepancy budget so that solutions that are increasingly different from the greedy solution are also explored. Demirović et al. (2023) follow the recursive search approach, but always expand unexpanded nodes before optimizing already expanded nodes. Their method also does not rely on caching, thus keeping its memory usage within a constant bound. Kiossou et al. (2025) combine several anytime approaches into a single *complete anytime beam search* (Zhang, 1998) adaptation of DL8.5. They present a generic approach that supports several heuristics to prune the search, e.g., based on information gain, node purity, search discrepancy, and exploring only the top-k features (following Blanc et al. (2023)).

2.3.7. Continuous Features

The algorithms discussed in this section so far have all assumed *binary* features. They assume that categorical and continuous features have, in some manner, been

binarized, possibly with loss of information. This binarization can be done either in an ad-hoc manner (e.g., one-hot encoding or based on minimum description length) or using a reference model to guess a good binarization (e.g., McTavish et al., 2022). In either case, the final tree will be optimal only for the given binarization.

Mazumder et al. (2022), on the other hand, propose Quant-BnB, a search algorithm that directly operates on continuous features and allows for all possible splits $[f \leq \tau]$ with f a feature and τ any threshold that yields a different split, i.e., the same splits that CART also considers.

Even for moderately sized datasets, Mazumder et al. show that this can result in several thousands to hundreds of thousands equivalent binary features, whereas the approaches above that assume binary features typically operate on less than a hundred binary features. E.g., Demirović et al. (2022) test on at most 1163 binary features, which is many more than previous approaches were tested on.

This large number of possible splits results in two problems: (i) the algorithms scale exponentially with respect to the number of possible splits because the amount of possible splits is the branching factor in the search algorithm; and (ii) MurTree's depth-two subroutine specifically becomes too expensive to fit in memory since it requires to store to store sample counts for each pair of features, i.e., quadratic memory in terms of number of possible splits.

Mazumder et al. (2022) approach this by splitting the search space by quantiles of the feature distribution. Solving the problem with splits only at those quantiles gives both upper and lower bounds for the original problem. For intervals of the feature distribution that cannot be pruned, more fine-grained quantiles of the feature distribution are used to recursively further tighten the gap, until the gap is closed and the optimal solution is found and proven.

2.3.8. Comparison

Having surveyed the main developments in DP and/or branch-and-bound based search for ODTs, this final subsection provides a brief qualitative and quantitative comparison of the state-of-the-art methods. Table 2.1 provides a qualitative overview, and we discuss this in more detail below.

DL8.5 (Aglin et al., 2020a) DL8.5 follows up on DL8 (Nijssen and Fromont, 2007, 2010) by expanding the DP search with branch-and-bound techniques. It implements depth-first search with binary features. Both dataset (bitset) and branch caching is supported. Hierarchical upper bounds help prune the search space. The tree size is bounded by a maximum depth. After publication, a variant of MurTree's depth-two solver was added to improve performance. Follow-up work extended DL8.5 to run within memory constraints (Aglin et al., 2022) and to improve anytime performance (Kiossou et al., 2022; Kiossou et al., 2025). The authors provide an accessible Python API that supports setting user-defined objectives (Aglin et al., 2020b).[5]

GOSDT (Lin et al., 2020) GOSDT improves their earlier OSDT approach (Hu

[5]The implementation of DL8.5 is available at `https://github.com/aia-uclouvain/pydl8.5`.

	DL8.5	GOSDT	MurTree	Quant-BnB	Branches
Caching techniques					
Branch	✓		✓		✓
Closure			✓		
Dataset (bitset)	✓	✓			
Dataset (list)			✓		
Bounding techniques					
Hierarchical	✓	✓	✓		
Similarity		✓	✓		
Sub-interval				✓	
Sparsity		✓			✓
Tree size bound					
Unbounded		✓			✓
Depth	✓	✓	✓	✓	✓
Nodes			✓		
Sparsity		✓	✓		✓
Search techniques					
Strategy	DFS	BFS	DFS	Quantiles	AND-OR
Features	Binary	Binary	Binary	Continuous	Categorical
Depth-two	✓*		✓	✓	

Table 2.1: Qualitative (simplified) comparison of state-of-the-art DP and branch-and-bound ODT methods. '*' means a feature was added after the original paper publication.

et al., 2019) by adding DP-based recursion and by generalizing its objective to any linear function of the false positives and negatives. It implements best-first search with a global priority queue. Its API supports continuous features but internally transforms them into binary features. A similarity bound is used to exploit similarities between binary features that originate from the same continuous feature. The primary method of bounding the tree size is by a sparsity penalty. The search space is pruned by a variety of bounds, several of which are based on this sparsity penalty. Follow-up work extended GOSDT to regression (Zhang et al., 2023) and survival analysis (Zhang et al., 2024). McTavish et al. (2022) extended GOSDT by "guessing" (possibly) suboptimal bounds and binarization from an ensemble reference model.[6]

MurTree (Demirović et al., 2022) MurTree implements a similar depth-first search as DL8.5 but extends it in several ways: it adds a depth-two subroutine and a similarity lower bound, and it allows to bound trees by a node budget in addition to a depth budget. This latter addition also enables optimizing trees by a sparsity penalty. It supports both branch, closure, and dataset (list) caching. Demirović and Stuckey (2021) extend MurTree for non-linear objectives such as F1-score. This thesis adapts and extends MurTree to a variety of objectives

[6]The implementation of GOSDT is available at `https://github.com/ubc-systopia/gosdt-guesses`.

and constraints.[7]

Quant-BnB (Mazumder et al., 2022) Quant-BnB's search strategy cannot be described by one of the core search paradigms. It splits the thresholds on continuous features into quantiles and splits on a subset of them. A variant of the similarity lower bound, the sub-interval bound is used to prune whole intervals of possible splits. Quant-BnB provides a subroutine specifically for solving depth-two trees and a routine for depth-three trees, but no recursive formulation. Quant-BnB can be used both for classification and regression.[8]

Branches (Chaouki et al., 2025) Branches uses the AND-OR search paradigm, with a priority queue for each search node, sorted by the lower bound of that search node. It primarily limits the tree size by a sparsity penalty and limiting by a maximum depth is also possible. Only branch caching is supported. Apart from binary features, Branches also supports categorical features.[9]

In order to give a brief quantitative comparison, we compare four of the five methods above on 23 UCI datasets (Dua and Graff, 2017): DL8.5, MurTree, GOSDT, and Branches. We compare all four without a sparsity penalty ($\lambda = 0$), and in addition report runtimes for GOSDT and Branches when $\lambda = 0.01|\mathcal{D}|$, since the performance of both these techniques depends on setting this parameter. The comparison trains depth-four trees on datasets with binarized features.[10] The comparison was run on an Intel Core Ultra 7 155U with 16 GB of RAM with a time out of 10 minutes.

Table 2.2 shows the results. When $\lambda = 0$, both DFS methods, DL8.5 and MurTree, stand out compared to the BFS and AND-OR approaches GOSDT and Branches. MurTree, on average, runs more than 200 times and almost 500 times faster than GOSDT and Branches respectively. On average, it also outperforms DL8.5 by a factor of five.[11]

By changing the sparsity penalty to $\lambda = 0.01|\mathcal{D}|$, the problem is changed and a comparison cannot directly be made. With the exception of one dataset (soybean) the problem becomes significantly easier to solve for both GOSDT and Branches. However, even in this simplified setting, their runtimes are still 62 and 174 times higher than MurTree.

For a comparison with Quant-BnB, we refer to the results reported by Mazumder et al. (2022). They compare (among others) with GOSDT, DL8.5, MurTree, and several MIP methods. For methods that require binarization, they create a binary feature for every possible split on the continuous data. They report that GOSDT, and the MIP methods do not finish within the four-hour time limit for any of the

[7] The implementation of MurTree is available at `https://github.com/MurTree/pymurtree`.
[8] The implementation of Quant-BnB is available at `https://github.com/mengxianglgal/Quant-BnB`.
[9] The implementation of Branches is available at `https://github.com/Chaoukia/branches`.
[10] The datasets were obtained from `https://bitbucket.org/EmirD/murtree-bi-objective`. Non-sparse features were flipped and duplicate features removed.
[11] The runtime performance between DL8.5 and GOSDT contradicts with what was observed by Lin et al. (2020). We see two reasons for that: (i) Lin et al. set $\lambda = 1/2^d$ to compare GOSDT with DL8.5, thinking that this would yield at most 2^d leaves for GOSDT, but this is not the case: it yields much smaller trees, such that GOSDT in their comparison is solving a much simpler problem than DL8.5; (ii) we test DL8.5 including the later added depth-two subroutine.

			$\lambda = 0$				$\lambda = 0.01\|\mathcal{D}\|$	
Dataset	$\|\mathcal{D}\|$	$\|\mathcal{F}\|$	DL8.5	MurTree	GOSDT	Branches	GOSDT	Branches
Primary-Tumor	336	17	< 1	< 1	< 1	< 1	< 1	< 1
Tic-Tac-Toe	958	27	< 1	< 1	2	7	2	5
Hepatitis	137	34	< 1	< 1	3	9	2	6
Kr-vs-kp	3196	38	< 1	< 1	9	22	6	11
Soybean	630	43	< 1	< 1	5	4	21	12
Hypothyroid	3247	44	< 1	< 1	13	29	< 1	1
Yeast	1484	46	< 1	< 1	15	42	11	35
Lymph	148	47	< 1	< 1	10	25	8	26
Vote	435	48	< 1	< 1	14	33	< 1	< 1
Anneal	812	49	< 1	< 1	6	35	5	24
Heart-Cleveland	296	56	< 1	< 1	28	90	16	78
Diabetes	768	56	2	< 1	36	118	21	104
Breast-Wisconsin	683	60	< 1	< 1	48	181	16	36
Australian-Credit	653	74	3	< 1	88	-	71	-
Audiology	216	84	< 1	< 1	215	-	153	-
German-Credit	1000	87	9	< 1	172	-	106	-
Mushroom	8124	100	1	< 1	-	-	-	-
Pendigits	7494	108	18	7	-	OoM	325	25
Letter	20000	112	49	16	-	OoM	12	162
Segment	2310	114	< 1	< 1	215	-	< 1	< 1
Vehicle	846	128	9	1	484	-	299	-
Ionosphere	351	222	131	12	-	-	-	-
Splice-1	3190	255	-	51	-	-	-	-
Average runtime ratio			5.4	1	201.0	489.4	62.4	173.6

Table 2.2: Runtime (s) for maximizing accuracy with maximum depth $d = 4$. Timeouts beyond 600s are denoted with '-'. Out of memory is indicated with 'OoM'. Average of five runs. The final row gives the geometric mean runtime ratio in comparison with MurTree (time-outs and out-of-memory are counted as 600s).

datasets (even when optimizing with a maximum depth of two). On most datasets, Quant-BnB outperforms DL8.5 and MurTree by a large margin, the latter two often running into memory problems. On a few datasets, MurTree still outperforms Quant-BnB. The out-of-sample accuracy results also confirm the benefit of splitting directly on the continuous features.

The good performance of MurTree motivates the further development of this approach in Chapters 3-6, under the assumption of binary features. In Chapter 8, we merge ideas from MurTree and Quant-BnB to improve performance on datasets with continuous features.

2.4. Model-Based Solving

Instead of custom search-based algorithms, the optimal decision tree problem is also solved by a variety of model-based solvers, such as mixed-integer programming

(MIP), continuous optimization (CO), constraint programming (CP), and Boolean satisfiability (SAT). In this thesis, we repeatedly compare our search-based methods with these model-based approaches, specifically MIP and SAT. Therefore, we introduce each of them here and briefly survey their corresponding literature.

2.4.1. Mixed-Integer Programming

Mixed-integer (linear) programming models problems as set of linear constraints and a linear objective. If all variables in the model are real-valued, the problem is called a linear program and it can be solved in polynomial time using interior-point optimization (Karmarkar, 1984). If, however, some of the variables are integers, the problem becomes NP-hard. Such problems are typically solved by relaxing the integrality constraints to get a simpler LP that can be solved to obtain a lower bound (in minimization problems). These bounds are then used in a branch-and-bound search where iteratively integer variables with non-integer assignments are fixed, until an integer feasible solution is found such that no open search node has an LP bound lower than the value of that solution.

Over the years, many different MIP models for solving optimal decision trees have been proposed. We here explain the most well-known variant, OCT proposed by Bertsimas and Dunn (2017), and then discuss how the other models deviate or improve upon OCT.

OCT Consider again a dataset $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}_{i=1}^{n}$, but now with $x^{(i)} \in [0,1]^{|\mathcal{F}|}$, i.e., all features are real-valued and normalized to the range $[0,1]$. Let d be the maximum depth, λ the complexity cost, and N_{min} the minimum required size of a leaf node. For a given maximum depth d, the number of nodes (branching and leaf nodes) in a binary tree equals $T = 2^{d+1} - 1$. Let $\mathcal{T}_B = \{1, ..., \lfloor T/2 \rfloor\}$ be the set of branching nodes and $\mathcal{T}_L = \{\lfloor T/2 \rfloor + 1, ..., T\}$ the set of leaf nodes. The parent of a node t is given by $p(t) = \lfloor t/2 \rfloor$ if $t > 1$. The set $A(t)$ is the set of ancestors of node t. This set is split into $A_L(t)$ and $A_R(t)$, the set of left and right branch ancestors, respectively. The left (right) branch ancestors are those ancestors for which the outgoing edge on the path toward t is the left (right) outgoing edge.

Splits in a node $t \in \mathcal{T}_B$ are computed as $a_t^T x \leq b_t$. Here, the variable $a_t \in \{0,1\}^{|\mathcal{F}|}$ captures the splitting variable of node t and b_t the splitting threshold. A node is considered 'off' if a_t is the all-zero vector. This is captured by the variable d_t. The following linear constraints capture this:

$$\sum_{j=1}^{|\mathcal{F}|} a_{jt} = d_t\,, \qquad \forall t \in \mathcal{T}_B\,, \tag{2.13}$$

$$0 \leq b_t \leq d_t\,, \qquad \forall t \in \mathcal{T}_B\,, \tag{2.14}$$

$$a_{jt} \in \{0,1\}\,, \qquad j \in [|\mathcal{F}|],\ \forall t \in \mathcal{T}_B\,, \tag{2.15}$$

$$d_t \in \{0,1\}\,, \qquad \forall t \in \mathcal{T}_B\,. \tag{2.16}$$

If a node is turned off ($d_t = 0$), then its children should also be turned off.

$$d_t \leq d_{p(t)}\,, \qquad \forall t \in \mathcal{T}_B \setminus \{1\}\,. \tag{2.17}$$

To keep track of which leaf each sample reaches, the variables $z_{it} = \mathbb{1}(x^{(i)}$ is in node $t)$ are introduced for each sample $i \in [n]$ and leaf node $t \in \mathcal{T}_L$. Whether sufficient samples reach a leaf is encoded by $l_t = \mathbb{1}(\text{leaf } t \text{ contains} \geq N_{min} \text{ samples})$. Each sample should end in precisely one leaf:

$$z_{it} \leq l_t\,, \qquad i \in [n],\ t \in \mathcal{T}_L\,, \tag{2.18}$$

$$\sum_{i=1}^{n} z_{it} \geq N_{min} l_t\,, \qquad t \in \mathcal{T}_L\,, \tag{2.19}$$

$$\sum_{t \in \mathcal{T}_L} z_{it} = 1\,, \qquad i \in [n]\,, \tag{2.20}$$

$$z_{it} \in \{0,1\}\,, \qquad i \in [n]\ \ \forall t \in \mathcal{T}_L\,, \tag{2.21}$$

$$l_t \in \{0,1\}\,, \qquad \forall t \in \mathcal{T}_L\,. \tag{2.22}$$

The model encodes in which leaf each sample ends up in, by examining the splits on the ancestors of each leaf node.

$$a_m^T(x^{(i)} + \varepsilon) \leq b_m + M_1(1 - z_{it})\,, \qquad i \in [n],\ \forall t \in \mathcal{T}_L,\ \forall m \in A_L(t)\,, \tag{2.23}$$

$$a_m^T x^{(i)} \qquad \geq b_m + M_2(1 - z_{it})\,, \qquad i \in [n],\ \forall t \in \mathcal{T}_L,\ \forall m \in A_R(t)\,. \tag{2.24}$$

In these last two equations, ε is a vector of small value to simulate a strict inequality. The values of the ε vector are equal to the minimal non-zero difference per feature between any two samples. The constants M_1 and M_2 are used to form big-M constraints. The smallest possible values to get a valid constraint are $M_1 = 1 + \max \varepsilon$ and $M_2 = 1$, since we have normalized values x.

To encode the objective, first a matrix Y is created as follows:

$$Y_{ik} = \begin{cases} +1\,, & \text{if } y^{(i)} = k \\ -1\,, & \text{otherwise} \end{cases}\,, \quad \forall k \in \mathcal{K},\ \ i \in [n]\,. \tag{2.25}$$

The number of samples in a leaf is counted by N_t and the number of samples with label k as N_{kt}.

$$N_{kt} = \frac{1}{2} \sum_{i=1}^{n} (1 + Y_{ik}) z_{it}\,, \qquad k \in \mathcal{K},\ t \in \mathcal{T}_L\,, \tag{2.26}$$

$$N_t = \sum_{i=1}^{n} z_{it}\,, \qquad t \in \mathcal{T}_L\,. \tag{2.27}$$

Let c_{kt} encode the predicted label in leaf t and L_t the number of misclassifications in leaf t. Then the following relations hold:

$$\sum_{k \in \mathcal{K}} c_{kt} = l_t, \qquad \forall t \in \mathcal{T}_L\,, \tag{2.28}$$

$$L_t \geq N_t - N_{kt} - M(1 - c_{kt}), \qquad k \in \mathcal{K},\ \forall t \in \mathcal{T}_L\,, \tag{2.29}$$

$$L_t \leq N_t - N_{kt} + M c_{kt}, \qquad k \in \mathcal{K},\ \forall t \in \mathcal{T}_L\,, \tag{2.30}$$

$$L_t \leq 0, \qquad \forall t \in \mathcal{T}_L\,. \tag{2.31}$$

Here, M is another big-M constant. Setting $M = n$ suffices.

The sum $\sum_{t \in \mathcal{T}_L} L_t$ now provides the total misclassification score, and combining this with the cost of adding nodes penalized by λ, the final objective is:

$$\min \quad \sum_{t \in \mathcal{T}_L} L_t + \lambda \sum_{t \in \mathcal{T}_B} d_t \,. \tag{2.32}$$

Oblique cuts By constraining each $a_{jt} \in \{0, 1\}$, the model above produces a decision tree with *axis-aligned* splits, the most common type of decision tree. By removing this constraint, the model can instead produce *oblique* (multi-variate) splits: splits that compute a linear combination of the feature values. Such oblique splits are more flexible, but are less human-comprehensible.

Recently, several papers propose SVM (support-vector machine) oblique splits (Zhu et al., 2020; Boutilier et al., 2023; Alès et al., 2024; D'Onofrio et al., 2024).

Developments After this, several studies have focused on improving the *scalability* of the MIP model. At the same time as Bertsimas and Dunn, Verwer and Zhang (2017) proposed an alternative MIP model with fewer decision variables. They followed up on this with another model that explicitly encodes all possible splits rather than doing so implicitly through continuous variables (Verwer and Zhang, 2019). Zhu et al. (2020) provide new cutting planes to remove the big-M constants, but also improve scalability through subsampling (hence losing optimality). Hua et al. (2022) reformulate the problem as a two-stage optimization problem, and deconstruct the problem into smaller subproblems that can be solved in parallel. Boutilier et al. (2023) also provide a stronger LP relaxation for multivariate decision trees and a Benders' decomposition that adds constrains for samples that cannot be separated by any multivariate split. D'Onofrio et al. (2024) introduce a quadratic formulation with fewer binary variables. Alès et al. (2024) also reformulate the problem as a quadratic problem, which, in turn, is used to obtain stronger LP relaxations.

Günlük et al. (2021) add splits for categorical features to the MIP model.

Aghaei et al. (2019) propose another model that also permits categorical variables. They focus on finding optimal models under a variety of fairness constraints. Aghaei et al. (2024) provide a new model based on a max-flow formulation which provides better scalability. The formulation has a stronger LP relaxation, and they exploit the max-flow / min-cut duality to solve the problem with Benders' decomposition. Jo et al. (2021), Justin et al. (2022), and Jo et al. (2023) use this model to learn prescriptive policy, robust, and fair optimal trees, respectively.

Finally, Aldinucci and Lapucci (2024) replace the zero-one loss with the logistic loss by using piecewise linear approximation.

Discussion Table 2.3 shows a comparison of three of the aforementioned MIP approaches: OCT (Bertsimas and Dunn, 2017), BinOCT (Verwer and Zhang, 2019), and FlowOCT (Aghaei et al., 2024) when training axis-aligned trees.[12] The comparison is done on the five smallest datasets of Table 2.2 with a maximum depth of three, a time-out of 600 seconds, and the same hardware. The models are solved

[12] Bo Tang provides an implementation of OCT, BinOCT, and FlowOCT at `https://github.com/LucasBoTang/Optimal_Classification_Trees`.

	Runtime (s) or MIP gap			Accuracy gap (%) versus optimal			
Dataset	OCT	BinOCT	FlowOCT	OCT	BinOCT	FlowOCT	CART
Primary-Tumor	(59%)	475	127	0	0	0	2.1
Tic-Tac-Toe	(100%)	(100%)	(26%)	2.0	1.3	1.9	2.1
Hepatitis	(88%)	(36%)	304	1.6	0	0	3.5
Kr-vs-kp	(100%)	(100%)	83	35.5	3.4	0	3.4
Soybean	(100%)	(96%)	59	2.9	0	0	2.9

Table 2.3: Comparison of three MIP approaches and CART, averaged over five runs, maximum depth $d = 3$. The left shows the runtime (s) of the MIP methods, or in case of a time out (600s), it reports in parentheses the MIP gap at time-out. The right shows the difference in training accuracy of the final solution to the optimal solution.

with Gurobi 12 (Gurobi Optimization LLC, 2022), and warm started with the CART solution (although the solver may ignore the warm start). The columns on the left report the runtime, if solved optimally, or the MIP gap at time out. The columns on the right show the gap between the training accuracy of the final returned solution and the optimal solution. For reference, the gap of CART's training accuracy to optimal is also included.

These results show that OCT did not finish within the time limit for any of the five datasets. It did find the optimal solution for Primary-Tumor, but the MIP gap at time out was still 59%. For Soybean, no improvement on the CART solution was found. For Tic-Tac-Toe the improvement on the CART solution was only 0.1%. For Kr-vs-kp, the final solution was much worse than the CART solution. BinOCT fares better by finishing one dataset within the time out, and finding optimal solutions for three datasets, although still with a large remaining MIP gap. For Kr-vs-kp, no improvement on the CART solution was found. FlowOCT has the best results among the three by finding and proving optimal solutions for four out of five datasets, and having the smallest remaining MIP gap for the fifth. FlowOCT therefore shows to be a considerable improvement over the original OCT model.

However, compared to the optimal DP approaches, the differences remain large, because these approaches solve the same problems in milliseconds, that is, four to five orders of magnitude faster. Possibly, this number is even higher since the experiments capped the runtime at 600 seconds. Therefore, despite improvements in scalability, the MIP approaches cannot yet compete with DP.

The comparison above was done on binarized data, since both BinOCT and FlowOCT expect explicitly setting all possible splits. OCT does not, so this indeed is an advantage. However, Mazumder et al. (2022) compare their branch-and-bound method, Quant-BnB, with OCT on datasets with continuous features and report that OCT did not finish for any dataset within the four-hour time limit, whereas both Quant-BnB and MurTree did.

One advantage of the MIP approach remains: it is easy to change to another (linear) objective or add a (global) linear constraint. Specifically, global constraints, such as fairness (Aghaei et al., 2019) or robustness (Vos and Verwer, 2022), are hard to enforce in a DP framework since DP assumes subtrees can be solved independently.

2.4.2. Continuous Optimization

Blanquero et al. (2021) turn the decision tree problem into a *non-linear continuous* optimization problem by changing the typical hard decision boundary of decision trees into a soft probabilistic decision boundary. Specifically, they model the decision boundary by a logistic cumulative distribution function (CDF). Feature vectors that are far from the decision boundary move with high probability to one branch of the split, whereas those close to the decision boundary go either way with similar probabilities. Similar to many MIP approaches above, they consider oblique splits. This approach outperforms the oblique version of OCT in scalability (and also accuracy, although this may be a result of time-outs). In later work, they improve sparsity of the splits by applying a l_1-norm (per split) and a l_∞-norm (globally) (Blanquero et al., 2020); and extend the approach to regression, for which they show that the sparsity improves the quality of local explanations (Blanquero et al., 2022).

Carrizosa et al. (2021) provide a survey of recent advances in both mixed-integer programming and continuous optimization for optimal decision tree learning.

2.4.3. Constraint Programming

Bessiere et al. (2009) use *constraint programming* (CP) to find decision trees of minimum size that perfectly classify the training data. They first explore the use of SAT (see Section 2.4.4), but the proposed SAT model does not scale as well as their CP approach. Their CP model does not fix the tree's topology, but only sets an upper bound on the number of nodes that can be used, so that the model can also find deep trees. They guide the search using the information gain heuristic.

Their search is further aided by a hitting-set lower bound. They create a set that records for each pair of samples with different labels, the features for which the pair has different values. The hitting set of this set ensures that each such pair is separated by at least one feature, and therefore provides a lower bound on the number of remaining nodes to add.

Verhaeghe et al. (2020) also use CP, but to maximize accuracy under a given depth constraint. They incorporate many of the same techniques also used in the DP approaches: caching, branch-and-bound, and a custom search that splits the problem into independent subproblems. They formalize their search strategy as AND-OR search, but in terms of terminology used above (see Section 2.3.4), their search method is best described as depth-first. When a branching decision is made, the CP state is updated accordingly. When the recursive call returns, the corresponding variable assignment is undone. If the state update propagates to an infeasible state, the CP solver returns immediately, and the search continues with the next branching assignment.

They show that the CP approach has a lower runtime than both the DP method DL8 (Nijssen and Fromont, 2007) and the MIP method BinOCT (Verwer and Zhang, 2019). However, in later comparisons, both DP methods DL8.5 (Aglin et al., 2020a) and MurTree (Demirović et al., 2022) significantly outperform the CP approach.

Therefore, at the moment, using CP for computing optimal decision trees is not competitive in scalability with DP. However, since global constraints on the tree

Year	Author(s)	Aim	Method	Features	Remark
2009	Bessiere et al.	Size	SAT	Binary	Also CP model
2018	Narodytska et al.	Size	SAT	Binary	
2020	Avellaneda	Depth, size	SAT	Binary	Incremental inference
2020	Hu et al.	Accuracy	MaxSAT	Binary	
2020	Janota and Morgado	Size	SAT	Binary	Enumerate topologies
2021	Shati et al.	Depth, accuracy	SAT, MaxSAT	Continuous, categorical	
2023	Alòs et al.	Size	MaxSAT	Binary	Finds multiple models
2025	Avellaneda	Accuracy	MaxSAT, MaxSMT	Continuous	Oblique splits

Table 2.4: An overview of SAT approaches for the optimal decision tree problem.

can easily be added to the CP model, future work could investigate, if CP could be beneficial in those scenarios.

2.4.4. Boolean Satisfiability

Several studies propose Boolean satisfiability (SAT) models to optimize decision trees. The previous section already mentioned Bessiere et al. (2009), who first propose a SAT model and then continue with a more scalable CP model. Their SAT model scales (in the best case) quadratically with respect to the number of samples.

Developments Narodytska et al. (2018) provide a new SAT approach that scales better because it is more tight, although it still requires a quadratic number of clauses with respect to the number of samples. Furthermore, they do not fix the tree topology. They run a linear search on the maximum number of nodes that permit a perfect classification tree, starting with an upper bound provided by a greedy heuristic solution.

Avellaneda (2020) improve scalability by incrementally adding misclassified training samples to the set of samples that is used to construct the SAT model. Moreover, they first find a perfect classification tree of minimum depth, then fix the depth, and minimize the number of tree nodes.

Hu et al. (2020) adapt the SAT model by Narodytska et al. to maximize accuracy under a given maximum size constraint by using MaxSAT. They also use their optimal decision trees as base learners in a boosted ensemble. In comparison to the DP approach DL8.5 their MaxSAT approach performs worse in runtime, but DL8.5 occasionally runs out of memory whereas their approach does not.

Janota and Morgado (2020) propose to reason about root-leaf *paths* rather than nodes, and allow to constrain both the tree size and the tree depth. Limiting the depth first, and then the number of nodes improves scalability. To further improve scalability, they split the SAT problem into two parts: (i) fixing the topology and (ii) labeling the tree. They enumerate topologies, fix the topology in the SAT model, and run the SAT solver for each.

Alòs et al. (2023) apply MaxSAT to find a perfect classification tree of minimum

size and extend the model by Narodytska et al. to multi-class classification. They use consecutive calls to the SAT solver to find a diverse set of equally-sized trees.

Up to this point, all proposed SAT models assumed *binary* features: categorical or continuous features are assumed to be encoded in binary form. Shati et al. (2021) overcome this limitation in their SAT model which directly splits on continuous features and allows *power-set* branching for categorical features. With power-set branching, a branching node can be labeled by any non-empty subset of the set of values a feature can take. If a sample has a value for that feature in the labeled set, it branches left, otherwise right. This type of branching is more flexible than only checking for equality of categorical features. In a follow-up publication (Shati et al., 2023b), they extend their model for cost-sensitive learning and include two types of tree pruning constraints: minimum support and a minimum margin constraint. The minimum margin constraint requires that any split, if applied to the whole dataset, would at least split off a minimum amount of samples. In comparison with previous SAT models, this new SAT model scales better, while also adding important improvements, such as branching on continuous and categorical features.

Avellaneda (2025) extends the SAT model by Shati et al. to permit oblique cuts with at most two non-zero weights. They solve this model either with Satisfiability Modulo Theories (SMT) or with MaxSAT where oblique cuts are encoded by reasoning over geometric relationships between all triples of three samples.

SAT model The following describes the SAT model by Shati et al. (2021). Let $a_{t,j}$ denote whether feature j is branched on in node t, then the following two constraints ensure that exactly one feature is chosen at each branching node $t \in \mathcal{T}_B$.

$$(\neg a_{t,f}, \neg a_{t,f'}), \qquad t \in \mathcal{T}_B, \quad f \neq f' \in \mathcal{F}; \tag{2.33}$$

$$\left(\bigvee_{f \in \mathcal{F}} a_{t,f} \right), \qquad t \in \mathcal{T}_B. \tag{2.34}$$

Let $u_f(i)$ denote the index of the i'th sample if the data $\mathcal{D}$ is sorted in ascending order by continuous feature f and let $O_f = \{(u_f(i), u_f(i+1)) \mid i \in [n-1]\}$ denote the set of all consecutive pairs by this ordering. Let $s_{i,t}$ denote that sample i goes left in branching node t, then continuous feature splits are encoded as follows:

$$(\neg a_{t,f}, s_{i,t}, \neg s_{i',t}), \qquad t \in \mathcal{T}_B, \quad f \in \mathcal{F}, \qquad (i,i') \in O_f; \tag{2.35}$$

$$(\neg a_{t,f}, \neg s_{i,t}, s_{i',t}), \qquad t \in \mathcal{T}_B, \quad f \in \mathcal{F}, \qquad (i,i') \in O_f, x_f^{(i)} = x_f^{(i')}. \tag{2.36}$$

As before in the OCT model, let $A_R(t)$ and $A_L(t)$ refer respectively to the set of right and left parents of node t, and let $z_{i,t}$ denote that sample i ends up in leaf t, then the following constraints ensure that each sample i follows a valid path to a

leaf t:

$$(\neg z_{i,t}, s_{i,t'}), \qquad t \in \mathcal{T}_L, \ i \in [n], \qquad t' \in A_L(t); \quad (2.37)$$

$$(\neg z_{i,t}, \neg s_{i,t'}), \qquad t \in \mathcal{T}_L, \ i \in [n], \qquad t' \in A_R(t); \quad (2.38)$$

$$\left(z_{i,t}, \bigvee_{t' \in A_L(t)} \neg s_{i,t'}, \bigvee_{t' \in A_R(t)} s_{i,t'}\right), \qquad t \in \mathcal{T}_L, \qquad i \in [n]. \quad (2.39)$$

Let $g_{t,c}$ denote whether leaf t gets label $c \in \mathcal{K}$, then ensure that at most one label can be selected:

$$(\neg g_{t,c}, \neg g_{t,c'}), \quad t \in \mathcal{T}_L, \quad c \neq c' \in \mathcal{K}. \quad (2.40)$$

Finally, we present here only the maximum accuracy case. Let p_i denote whether sample i is classified correctly in the tree, and add (p_i) for each sample i as a soft clause. The value of p_i is decided by the constraint that checks that sample i obtains the label it should obtain:

$$(\neg p_i, \neg z_{i,t}, g_{t,y^{(i)}}), \quad t \in \mathcal{T}_L, \quad i \in [n]. \quad (2.41)$$

Shati et al. furthermore add constraints that ensure that samples with the lowest (highest) feature value for a splitting feature f are sent left (right), and constraints for branching on categorical features. In their later work (Shati et al., 2023b), they further expand on the model as explained above. Let n be the number of samples, m the number of features, s the maximum tree size, and k the number of classes, then, as can be observed by going over all equations above, the final model has $\mathcal{O}(m^2 + nms + k^2)$ clauses.

Discussion The first SAT approach lacked scalability, but recent advances have significantly improved upon this. Shati et al. (2023b) compare with the DP approach DL8.5 (binarizing continuous features for DL8.5), and observe that on larger datasets (those where the binarization yields many binary features), their SAT approach scales better than DL8.5. On binary datasets, DL8.5 still clearly outperforms the SAT approach.

As with other model-based approaches, it is relatively easy to add global constraints to the model, provided they can be encoded compactly with Boolean logic.

2.5. Applications

The sections above highlighted research into optimal *classification* trees with either the standard accuracy or the minimum size objective (in the case of SAT). This section broadens the view to other applications and objectives. The aim is not to provide an exhaustive literature review of ODTs for these applications. We provide those in the respective chapters, e.g., on regression in Chapter 6 and on survival analysis in Chapter 5. The aim is to highlight general observations and challenges.

Methods Table 2.5 provides an overview of papers that study application or adaptation of ODTs to other objectives and constraints. We leave out papers that

Year	Author(s)	Method	Application / objective
2017	Bertsimas et al.	MIP	Regression
2022	Blanquero et al.	CO	Regression
2023	Zhang et al.	DP	Regression
2017	Kallus	MIP	Prescriptive policies
2019	Bertsimas et al.	MIP	Prescriptive policies
2021	Jo et al.	MIP	Prescriptive policies
2022	Zhou et al.	Search	Prescriptive policies
2022	Amram et al.	MIP	Prescriptive policies
2019	Aghaei et al.	MIP	Fairness
2023	Jo et al.	MIP	Fairness
2021	Vilas Boas et al.	MIP	Algorithm selection
2021	Demirović and Stuckey	DP	Nonlinear classification metrics
2022	Vos and Verwer	MIP	Robustness
2022	Justin et al.	MIP	Robustness
2023	Vos and Verwer	MIP	MDP policies
2024	Lemaire et al.	DP	Quantile regression
2024	Zhang et al.	DP	Survival analysis
2025	Staus et al.	Search	Perfect classification
2025	Shati et al.	SAT	Clustering

Table 2.5: Overview of research into optimal decision trees for other applications and objectives.

have been discussed before (such as Verwer and Zhang (2017), who also discuss regression and fairness). From the table, two observations stand out: the prevalence of MIP approaches and the absence of SAT and CP approaches.

Despite the scalability challenge, the majority of papers in Table 2.5 use MIP. With MIP, it is easy to change the objective or add global linear constraints, such as a demographic parity constraint (Jo et al., 2023). However, many of these works do not consider, interact with, or compare with the broader literature beyond MIP.

While both CP and SAT are also general-purpose model-based solvers, we do not observe a broader application of these methods, even though they potentially could. Shati et al. (2023b), for example, use MaxSAT for cost-sensitive classification, and this, with some small changes, could be applied to prescriptive policy generation or algorithm selection.

Global objectives and constraints Nanfack et al. (2022) point out that one of the challenges for decision tree algorithms is *global* constraints or objectives, i.e., those that cannot be optimized independently in subtrees. Most greedy approaches split up the problem into independent subproblems and therefore cannot easily handle global constraints such as fairness or robustness.

In contrast, the MIP, SAT, and CP approach can easily incorporate such global

constraints, and this is one of the important advantages of optimal decision trees over greedy approaches. However, these methods cannot fully exploit the recursive (separable) property of the tree structure.

Dynamic programming seems to have the same limitation as the greedy approaches here: it exploits the tree structure to solve subtrees as independent subproblems, and as such obtains better scalability than SAT, CP, and MIP. However, it is more difficult to add global constraints and objectives.

Demirović and Stuckey (2021) are an exception to this rule. They optimize nonlinear metrics, such as F1-score (the harmonic mean of precision and recall), that cannot be independently optimized in subtrees. They solve this by keeping track of Pareto front of nondominated subtree solutions that might be part of a global optimal solution. Starting from this work, this thesis continues to explore the application of DP to such global objectives and constraints in Chapters 3 and 4.

3

Fair and Optimal Decision Trees

Preface This chapter shows how a global constraint such as group fairness that introduces dependencies between the optimal solutions of left and right subtrees, can still be solved using dynamic programming. The method introduced in this chapter forms a stepping stone to the general framework we present in Chapter 4.

Abstract Interpretable and fair machine learning models are required for many applications, such as credit assessment and in criminal justice. However, state-of-the-art algorithms for fair and optimal decision trees have scalability issues, often requiring several hours to find such trees even for small datasets. Previous research has shown that dynamic programming (DP) performs well for optimizing decision trees because it can exploit the tree structure. However, adding a global fairness constraint to a DP approach is not straightforward, because the global constraint violates the condition that subproblems should be independent. We show how such a constraint can be incorporated by introducing upper and lower bounds on final fairness values for partial solutions of subproblems, which enables early comparison and pruning. Our results show that our model can find fair and optimal trees several orders of magnitude faster than previous methods, and now also for larger datasets that were previously beyond reach. Moreover, we show that with this substantial improvement our method can find the full Pareto front in the trade-off between accuracy and fairness.

3.1. Introduction

As machine learning (ML) is used in more domains that involve discrimination-sensitive decisions, demand for *fair*, *interpretable* and *accurate* models increases. ML, for example, is used in criminal justice (Berk et al., 2021), credit assessment (Kozodoi et al., 2022), and housing appointments (Azizi et al., 2018), each of which requires

Parts of this chapter have been published in Van der Linden, De Weerdt, and Demirović (2022), "Fair and Optimal Decision Trees: A Dynamic Programming Approach", in *Advances in NeurIPS-22*.

fair decisions. Simply removing the discrimination-sensitive attribute from the dataset does not necessarily result in less discrimination (Pedreschi et al., 2008). Instead, fairness is obtained by adding a constraint on the classifier: often either an individual or a group fairness constraint (see Dwork et al. (2012) and Binns (2020) for a comparison of the two). The focus in this chapter is on *group fairness*. With group fairness, a binary classifier is considered fair if the probability of receiving the positive outcome is the same for all protected groups. This fairness metric is formulated as a global constraint over the full dataset.

There are several approaches for finding fair and optimal decision trees (Verwer and Zhang, 2017; Aghaei et al., 2019; Jo et al., 2023; Aghaei et al., 2024), all of which use mixed-integer programming (MIP). However, none of these models scale well. If fairness is not considered, small optimal decision trees can be found efficiently by using dynamic programming (DP) methods that exploit the separability present in the tree structure (Hu et al., 2019; Aglin et al., 2020a,b; Lin et al., 2020; Demirović et al., 2022). Demirović and Stuckey (2021) also use DP to find the Pareto front of optimal decision trees for biobjective nonlinear metrics.

However, a global fairness constraint cannot trivially be added to these algorithms, because the fairness constraint breaks the separability assumption by introducing dependency between subproblems. When optimizing for accuracy, for example, left and right subtrees can be solved independently after choosing what feature to branch on in the current decision node. In contrast, when optimizing for fairness, the left and right subtree cannot be optimized independently, since imbalance in one part of the tree can be cancelled out in another part of the tree.

Motivated by the success of DP methods for conventional optimal trees, we developed a novel DP method that successfully incorporates a global fairness constraint. We solve the subproblem dependency by computing upper and lower bounds on the final fairness values of partial solutions, thus enabling early pruning of the search space. Moreover, we present algorithmic improvements in the merging and comparing of subproblems, and perform a study of scalability. Overall our approach provides orders of magnitude improvements over state-of-the-art MIP methods.

This has several benefits. First, our algorithm can find fair and optimal decision trees for datasets that were previously too large to consider, often even within seconds. Second, it can find deeper and more accurate trees, while still satisfying the fairness constraint. Last, our algorithm can find the full Pareto-front in the trade-off between accuracy and fairness. This enables the responsible domain expert to make a well-informed decision and select a policy from the solution set that fits the context and application best.

As is commonly done in optimal decision tree search, we assume that all features have been binarized in advance (Carrizosa et al., 2021). Similarly, it is assumed that the label of every instance in the dataset is binary: either the positive/preferred outcome or the negative outcome. This is a common assumption in fair classification (Denis et al., 2021) and was also assumed in previous fair and optimal decision tree methods (Verwer and Zhang, 2017; Jo et al., 2023).

The chapter setup is as follows: we first discuss related work and preliminaries, and then we present our method. Finally, we evaluate the performance of the new

method on twelve datasets from different application domains and we compare its performance to a leading MIP method from the literature.

3.2. Related Work

This section discusses the related work (i) MIP models for finding fair optimal decision trees and (ii) other (non-optimal) methods for finding fair decision trees. For related work on optimal decision trees in general, see Chapter 2.

Fairness in optimal decision trees Currently, the only methods for finding fair and optimal decision trees are MIP-based. Verwer and Zhang (2017) developed a MIP model with a soft group fairness constraint in the objective. They report how their method can find a fair depth-three tree for a dataset of 1000 instances in 15 minutes, when the model is given an initial solution from CART. But for the fair tree case they do not report being able to prove optimality within this time limit.

Aghaei et al. (2019) present a MIP model that co-optimizes fairness and error for both classification and regression. Apart from group fairness, they also consider a notion of individual fairness separately. However, they are criticized by Ranzato et al. (2021) that their model often provides accuracy not much better than a constant classifier. Jo et al. (2023) and Aghaei et al. (2024) continue their work and present a MIP model based on a max-flow formulation that results in a considerable improvement of the runtime. However, these models still have scalability issues because of the large number of binary variables.

Other methods for fair decision trees For completeness the following section mentions a number of related fair decision tree methods that do not guarantee a globally optimal solution. These can be divided into pre-, in-, and postprocessing methods.

Kamiran and Calders (2009) present a preprocessing method that changes the labels in the dataset according to a group fairness metric. The updated dataset can then be used to train any regular classifier.

Some of the in-processing methods consider fairness in the classifier itself. Others use a regular classifier, combined with a meta- or master problem that takes care of the fairness constraint. Pedreschi et al. (2008), for example, develop a method for finding fair association rules that can then be used to compile decision trees. Agarwal et al. (2018) iteratively solve a number of cost-sensitive classification problems, with the costs determined by the number of violations of the fairness constraint. Grari et al. (2020) propose a gradient tree boosting approach with an adversarial fairness constraint. Detassis et al. (2021) use a MIP master problem that changes the labels for the next training iteration to satisfy the fairness constraint. There is, however, no guarantee of convergence. Valdivia et al. (2021) developed an evolutionary algorithm that returns a Pareto front of decision trees with the multiobjective of both fairness and accuracy.

Kamiran et al. (2010) test a CART-based approach with a split criterion that is a mix of information gain in the label and the sensitive feature. They did not observe a significant improvement when using this approach. As an alternative, they suggest

a postprocessing relabeling method that uses a greedy knapsack algorithm to decide which leaf nodes to relabel to improve fairness at a minimum cost of accuracy.

Summary of related work In summary, the current state of the art has scalable custom methods for finding optimal conventional decision trees, but adding fairness constraints to these methods is non-trivial. MIP-based solutions exist for this problem, but they do not scale well. The alternative is to use heuristic methods, which result in suboptimal solutions.

3.3. Preliminaries

This section discusses the preliminaries of our method: an introduction of notation, a formal introduction of group fairness, a formal problem definition, and a dynamic programming formulation for optimal decision tree search that does not consider fairness.

Notation A dataset $\mathcal{D}$ contains instances $(\mathbf{x}, a, y)$, with $\mathbf{x}$ the feature vector, a the sensitive attribute and y the label value. The number of instances in the dataset $\mathcal{D}$ is $N = |\mathcal{D}|$. Let y_i be the label of instance i and a_i the value of the protected feature. Let $\mathcal{F}$ be the feature set and $f \in \mathcal{F}$ a feature in the feature set. We use f_i to denote the value of this feature for instance i. Since we are only considering binary features and binary classification, we introduce the shorter notation f_i and $\bar{f}_i$ to mean respectively that instance i satisfies or does not satisfy feature f. The same idea applies for y and $\bar{y}$, and a and $\bar{a}$. We use subscript notation to refer to a subgroup of the dataset: for example, $\mathcal{D}_{\bar{f}}$ refers to all instances in dataset $\mathcal{D}$ that do not satisfy feature f; N_a is the number of instances in the dataset that belong to group a, and $N_{y,\bar{a}}$ is the number of instances in the dataset with positive outcome that does not belong to group a.

Group fairness formalized The specific group fairness definition considered in this chapter is *demographic parity*, although our method can easily be extended to support other notions of group fairness, such as equality of opportunity. Demographic parity requires that the performance (the expected percentage of positive outcomes) per group is the same (Dwork et al., 2012). More formally, if $\hat{y}$ is the predicted outcome, 1 is the positive outcome, and a represents a (binary) sensitive/protected group, then demographic parity holds iff

$$P(\hat{y} = 1 \mid a = 1) = P(\hat{y} = 1 \mid a = 0)\,. \tag{3.1}$$

In binary classification, this constraint can be measured by considering the relative average performance of the two groups. Let $\hat{p}$ be the percentage of instances that receive the positive outcome, so $\hat{p}_a = N_{\hat{y},a}/N_a$. The difference in the performance between two groups is what we will call the imbalance $I = \hat{p}_a - \hat{p}_{\bar{a}}$. When the imbalance is limited to some value δ, we say that it satisfies demographic parity up to bias δ:

$$|I| = |\hat{p}_a - \hat{p}_{\bar{a}}| \leq \delta\,. \tag{3.2}$$

According to the definition of demographic parity, if a leaf node n receives label 0, the partial imbalance I_n in that node is also 0. Otherwise, the partial imbalance in

a leaf node n can be expressed as follows:

$$I_n = \frac{N_{n,a}}{N_a} - \frac{N_{n,\bar{a}}}{N_{\bar{a}}} \,. \tag{3.3}$$

The partial imbalance of a branching node is the sum of the imbalance of both sub-nodes.

Problem definition The task of learning an optimal fair decision tree is to find the feature value tests in the branch nodes and the leaf node assignments for a tree of a given maximum size that minimize the number of misclassifications in a training dataset while observing a fairness constraint. The percentage of correctly classified instances is called accuracy. This can be formalized as follows. The task is to find a decision tree classifier $h : \{0,1\}^{|\mathcal{F}|} \to \{0,1\}$ of depth d that minimizes the misclassification score for a given dataset $\mathcal{D}$ while observing a fairness constraint up to bias δ:

$$\begin{aligned} &\min_h \sum_{(\mathbf{x},a,y)\in\mathcal{D}} |h(\mathbf{x}) - y| \\ &\text{s.t.} \left| \sum_{(\mathbf{x},a,y)\in\mathcal{D}_a} \frac{h(\mathbf{x})}{N_a} - \sum_{(\mathbf{x},a,y)\in\mathcal{D}_{\bar{a}}} \frac{h(\mathbf{x})}{N_{\bar{a}}} \right| \leq \delta \,. \end{aligned} \tag{3.4}$$

Alternatively, when searching for the full Pareto front, the parameter δ is not necessary, and the problem can be written as a multiobjective problem:

$$\min_h \left\{ \sum_{(\mathbf{x},a,y)\in\mathcal{D}} |h(\mathbf{x}) - y|, \left| \sum_{(\mathbf{x},a,y)\in\mathcal{D}_a} \frac{h(\mathbf{x})}{N_a} - \sum_{(\mathbf{x},a,y)\in\mathcal{D}_{\bar{a}}} \frac{h(\mathbf{x})}{N_{\bar{a}}} \right| \right\} . \tag{3.5}$$

The Pareto front for Eq. (3.5) consists of all nondominated solutions (M, I) with M the misclassification score and I the imbalance. A solution is considered dominated if there is another solution that performs better or similar on all objectives. Therefore, we can define the dominance relation ($\succ$) for two solutions (M_1, I_1) and (M_2, I_2), and the function nondom as follows:

$$(M_1, I_1) \succ (M_2, I_2) \text{ iff } M_1 \leq M_2 \wedge |I_1| \leq |I_2| \wedge (M_1, I_1) \neq (M_2, I_2) \,, \tag{3.6}$$

$$\text{nondom}(S) = \{s_1 \in S \mid \neg\exists\ s_2 \in S, s_2 \succ s_1\} \,. \tag{3.7}$$

Dynamic programming formulation The problem of finding accurate decision trees has been formulated as a DP problem before by Demirović et al. (2022). Of interest to this chapter is the formulation for biobjective optimization by Demirović and Stuckey (2021). They minimize the misclassification score of two classes at the same time and return the Pareto front of nondominated solutions:

$$T_{\text{BI}}(\mathcal{D}, d) = \begin{cases} \{(0, |\mathcal{D}_y|), (|\mathcal{D}_{\bar{y}}|, 0)\} & d = 0 \\ \text{nondom}\left(\cup_{f\in\mathcal{F}} \text{merge}\left(T_{\text{BI}}(\mathcal{D}_f, d-1), T_{\text{BI}}(\mathcal{D}_{\bar{f}}, d-1)\right)\right) & \text{else} \end{cases} \tag{3.8}$$

At the leaf node ($d = 0$), the function returns two solutions consisting of two values: the misclassification score per class (e.g., $(0, |D_y|)$ when label $\hat{y} = 0$ is selected, because all instances with $y = 0$ will be correctly classified and all instances with $y = 1$ will cause $|D_y|$ misclassifications for class 1). At a branching node ($d > 0$), the function should return the merge of two solution sets and retain only the nondominated solutions resulting from that merge. The function merge combines two sets of partial solutions (from the left and right subtree) by element-wise addition:

$$\text{merge}(S_1, S_2) = \{(a_1 + a_2, b_1 + b_2) \mid (a_1, b_1) \in S_1, (a_2, b_2) \in S_2\} \,. \tag{3.9}$$

3.4. Method Description

In this section, we present our method DPF (DP Fair) and show how to deal with a global fairness constraint that introduces subproblem dependency. Similar to Eq. (3.8), we define a recursive function T_{F} that returns a set of all nondominated (partial) solutions, resulting in a full Pareto front of fair and optimal decision trees. In order to guarantee that the method returns the full Pareto front we must do an exhaustive search of all combinations of subtrees, pruning only those solutions for which we can prove that they will never be part of an optimal (nondominated) solution.

Dependency The recursive function Eq. (3.8) introduced in the preliminaries assumes that the left and the right tree can be independently optimized, but this is not the case when considering group fairness, which is a global constraint on the whole tree. When comparing two possible subtrees, one of which discriminates against group A and the other discriminating against group B, it is not clear which one is better, because it will depend on what happens in the rest of the tree. Even when both subtrees discriminate against the same group A, it is not immediately clear that the one with lower discrimination score will be better, but again it will depend on what happens in the rest of the tree.

More formally, the previous proposed biobjective method requires objectives to be *additive* and *monotonic* in order to create independent subproblem. The additive property holds if two partial solutions (a, b) and (a', b') can be combined by addition: $(a + a', b + b')$. The monotonic property holds for a biobjective function $f(a, b)$ if for any two possible different inputs a, b and a', b', the following dominance relation holds:

$$a \leq a' \wedge b \leq b' \rightarrow f(a, b) \succ f(a', b') \,. \tag{3.10}$$

However, because fairness is measured by the absolute value of the imbalance, both the additive property and the monotonic property are not satisfied. Our method addresses this problem.

Upper and lower bounds for fairness The key idea in our method is to compute upper and lower bounds for the imbalance value of partial solutions to enable comparison. When upper and lower bounds $\bar{I}_R$ and $\underline{I}_R$ are known for the imbalance in the rest (R) of the tree, then the final imbalance value of a tree that contains node n will be in the range: $[\underline{I}_R + I_n, \bar{I}_R + I_n]$. The lower bound ($\underline{I}$) and

the upper bound ($\bar{I}$) for the final *absolute* imbalance value can now be computed:

$$\underline{I}(I_n, [\underline{I}_R, \bar{I}_R]) = \begin{cases} 0 & \text{if } 0 \in [\underline{I}_R + I_n, \bar{I}_R + I_n] \\ \min\left(|\underline{I}_R + I_n|, |\bar{I}_R + I_n|\right) & \text{else} \end{cases} \tag{3.11}$$

$$\bar{I}(I_n, [\underline{I}_R, \bar{I}_R]) = \max\left(|\underline{I}_R + I_n|, |\bar{I}_R + I_n|\right) \tag{3.12}$$

In Eq. (3.11), the lower bound is zero if the permitted range contains zero; otherwise it is the minimum absolute value in that range. In Eq. (3.12), the upper bound is the maximum absolute value in the permitted range.

With these upper and lower bounds on the final absolute imbalance value, we can redefine the dominance relation of Eq. (3.6):

$$\begin{aligned} (M_1, I_1) \succ (M_2, I_2)) \text{ iff } & (M_1, I_1) \neq (M_2, I_2) \\ & \wedge\ M_1 \leq M_2 \\ & \wedge\ \left(\bar{I}(I_1, [\underline{I}_R, \bar{I}_R]) \leq \underline{I}(I_2, [\underline{I}_R, \bar{I}_R])\ \vee\ I_1 = I_2\right) . \end{aligned} \tag{3.13}$$

A solution is dominated by another solution if 1) the other solution has a lower misclassification score; and 2) either has an upper bound on the absolute imbalance lower than the lower bound on the absolute imbalance of this solution, or has precisely the same imbalance score.

Finding the Pareto front of optimal fair trees Now we can define a new function for finding fair trees $T_F(\mathcal{D}, d, [\underline{I}_R, \bar{I}_R])$ that optimizes Eq. (3.5):

$$T_{\mathrm{F}}(\mathcal{D}, d, [\underline{I}_R, \bar{I}_R]) = \begin{cases} \mathrm{leaf}(\mathcal{D}, [\underline{I}_R, \bar{I}_R]) & d = 0 \\ \mathrm{branch}(\mathcal{D}, d, [\underline{I}_R, \bar{I}_R]) & \text{else} \end{cases} \tag{3.14}$$

The functions leaf and branch will be introduced next. Both use the nondom function, which is now based on our new dominance comparison as described in Eq. (3.13) (but for ease of notation, $[\underline{I}_R, \bar{I}_R]$ will often be left out).

In a leaf node n, the function should return the following (M, I) solutions, with imbalance values based on Eq. (3.3):

$$\mathrm{leaf}(\mathcal{D}, [\underline{I}_R, \bar{I}_R]) = \mathrm{nondom}\left(\left\{(|\mathcal{D}_y|, 0), \left(|\mathcal{D}_{\bar{y}}|, \frac{N_{n,a}}{N_a} - \frac{N_{n,\bar{a}}}{N_{\bar{a}}}\right)\right\}\right) . \tag{3.15}$$

Eq. (3.15) returns up to two solutions, each described by two values: first the misclassification score as before, and second the imbalance when assigning the positive outcome.

In a branching node, the dataset is split on some feature f, resulting in two subproblems. Each subproblem is solved independently by passing the best known bounds for the other subproblem. Let $U(\mathcal{D}_{\bar{f}}, [\underline{I}_R, \bar{I}_R], d-1)$ denote the function that returns these best known bounds. These bounds can be based on dataset inspection, or based on previous (cached) solutions. Bounds from inspecting the dataset can be derived by assigning one label to all instances of one group and the other label to all other instances and vice versa. The results provide the maximum discrimination that

could possibly happen. When (cached) partial solutions for the rest of the tree are known, the bounds can be derived by taking the minimum and maximum imbalance values among solutions. With these bounds, the solutions from the two subproblems are then merged as follows:

$$\begin{aligned}\text{branch}(\mathcal{D}, d, [\underline{I}_R, \bar{I}_R]) = \text{nondom}\,(\cup_{f\in\mathcal{F}} \text{merge}\,(&\\ T_F(\mathcal{D}_{\bar{f}}, d-1, \quad U(\mathcal{D}_f, [\underline{I}_R, \bar{I}_R], d-1)),&\\ T_F(\mathcal{D}_f, d-1, \quad U(\mathcal{D}_{\bar{f}}, [\underline{I}_R, \bar{I}_R], d-1)))\big)\ .&\end{aligned} \tag{3.16}$$

Finding a single best tree It is also possible with this same DP formulation to search for a single optimal and fair tree up to bias δ, as defined in Eq. (3.4). This can be achieved by pruning all (partial) solutions and retaining only those that are *possibly* fair. Then, in the root node of the search, the tree with minimum misclassification score is selected from the list of solutions. A solution is possibly fair if the lower bound on the absolute imbalance value is less than or equal to δ. For this, we introduce the function prune, which should filter a set of solutions S before it is passed to the nondom function:

$$\text{prune}(S, [\underline{I}_R, \bar{I}_R]) = \left\{(M_n, I_n) \in S \mid \ \underline{I}(I_n, [\underline{I}_R, \bar{I}_R]) \leq \delta\right\}\ . \tag{3.17}$$

Pseudocode This section gives the pseudocode for DPF. In addition to the description above, the pseudocode also considers upper and lower bounds and cache. Therefore, the function T_F receives the current best known upper bound ub on the misclassification score as an extra parameter.

The function prune must be redefined to take into account this upper bound on the misclassification score:

$$\text{prune}(S, \text{ub}, [\underline{I}_R, \bar{I}_R]) = \left\{(M_n, I_n) \in S \mid \ \underline{I}(I_n, [\underline{I}_R, \bar{I}_R]) \leq \delta, M_n < \text{ub}\right\}\ . \tag{3.18}$$

For shorter notation, define

$$\text{filter}(S, \text{ub}, [\underline{I}_R, \bar{I}_R]) = \text{nondom}\left(\text{prune}\left(S, \text{ub}, [\underline{I}_R, \bar{I}_R]\right), [\underline{I}_R, \bar{I}_R]\right)\ . \tag{3.19}$$

Also we define the functions $\underline{M}$ to return the best misclassification score within a set of solutions:

$$\underline{M}(S) = \min\left\{M \mid (M, I) \in S\right\}. \tag{3.20}$$

Furthermore, let LB return the best known lower bound on the misclassification score for a subproblem, or zero if no such lower bound is known.

With these changes, see Algorithm 5 for the resulting pseudocode of DPF. In this algorithm S is the set of solutions, and lb and ub are the lower and upper bounds, with subscripts L and R denoting left and right subtrees. For a full Pareto front, the algorithm must be called with a fairness cut-off value of $\delta = 1$.

Other algorithmic improvements There are a number of other improvements in DPF for reducing the runtime. To reduce the runtime cost of the merge function, before merging, the two sets of partial solutions are first pruned based on bounds

Algorithm 5: Tree search of depth d with fairness on a dataset $\mathcal{D}$ for a feature set $\mathcal{F}$.

$T_{\mathrm{F}}(\mathcal{D}, d, \mathrm{ub}, [\underline{I}_R, \bar{I}_R])$
  if $d = 0$ **then**
    $S \leftarrow \left\{(|\mathcal{D}_y|, 0), \left(|\mathcal{D}_{\bar{y}}|, \frac{N_{n,a}}{N_a} - \frac{N_{n,\bar{a}}}{N_{\bar{a}}}\right)\right\}$
    return $\mathrm{filter}(S, \mathrm{ub}, [\underline{I}_R, \bar{I}_R])$
  $\langle S, \mathrm{lb}, \mathrm{stat}\rangle \leftarrow \mathrm{cache}[\mathcal{D}, d, [\underline{I}_R, \bar{I}_R]]$
  if $\mathrm{lb} \geq \mathrm{ub}$ **then return** $\emptyset$
  if $\mathrm{stat} = \mathrm{optimal}$ **then return** $\mathrm{filter}(S, \mathrm{ub}, [\underline{I}_R, \bar{I}_R])$
  $S \leftarrow \emptyset$
  for $f \in \mathcal{F}$ **do**
    $\mathrm{lb}_R \leftarrow \mathrm{LB}(\mathcal{D}_f, d-1, [\underline{I}_R, \bar{I}_R])$
    $S_L \leftarrow T_{\mathrm{F}}(\mathcal{D}_{\overline{f}}, d-1, \mathrm{ub} - \mathrm{lb}_R, U(\mathcal{D}_f, [\underline{I}_R, \bar{I}_R], d-1))$
    if $S_L = \emptyset$ **then continue**
    $\mathrm{lb}_L \leftarrow \mathrm{LB}(\mathcal{D}_{\overline{f}}, d-1, [\underline{I}_R, \bar{I}_R])$
    $S_R \leftarrow T_{\mathrm{F}}(\mathcal{D}_f, d-1, \mathrm{ub} - \mathrm{lb}_L, U(\mathcal{D}_f, [\underline{I}_R, \bar{I}_R], d-1))$
    if $S_R = \emptyset$ **then continue**
    $S \leftarrow S \cup \mathrm{prune}\left(\mathrm{merge}(S_L, S_R, [\underline{I}_R, \bar{I}_R]), \mathrm{ub}, [\underline{I}_R, \bar{I}_R]\right)$
  $S \leftarrow \mathrm{nondom}(S)$
  if $S = \emptyset$ **then**
    $\mathrm{cache}[\mathcal{D}, d, [\underline{I}_R, \bar{I}_R]] \leftarrow \langle \emptyset, \mathrm{ub}, \text{lower bound}\rangle$
    return $\emptyset$
  $\mathrm{cache}[\mathcal{D}, d, [\underline{I}_R, \bar{I}_R]] \leftarrow \langle S, \underline{M}(S), \mathrm{optimal}\rangle$
  return S

on the misclassification score and discrimination score derived from the other set. Then the two sets are sorted by misclassification score to enable early termination of the merge if it can be proven that all consecutive combinations of partial solutions will exceed the misclassification upper bound. In the root node of the search, the solutions are sorted on the imbalance value to enable faster elimination of non-fair solutions.

The runtime of the nondom function can be reduced by observing that partial solutions with a lower bound imbalance of 0 do not need to be compared with other solutions because they will always be nondominated. For the sake of brevity, all these have been left out of the method description. See our code repository for full implementation details.[1]

Similar to Demirović and Stuckey (2021) and Demirović et al. (2022) our method also uses a special depth-two solver to reduce the runtime. Our source code contains the full details. Appendix 3.A presents a complexity analysis of DPF. Appendix 3.B shows how our algorithm can easily be modified to also optimize tree sparsity.

[1] `https://doi.org/10.4121/3584f9cd-cf7b-478c-960f-3b2ef47d1427`

3.5. Experimental Results

The following section analyzes the performance of DPF. We focus on the analysis of the runtime and scalability of DPF: 1) How does DPF compare to the state of the art in terms of runtime? 2) What impact do our pruning method and merge improvements have on the runtime? 3) What is the runtime cost to find the full Pareto front? 4) What impact do parameters such as number of features, instances, etc. have on the runtime of DPF? For an out-of-sample analysis, see Appendix 3.D.

Experiment setup We evaluate DPF on all datasets mentioned in the survey by Le Quy et al. (2022). The data preprocessing and binarization is also done as described in that paper. Categorical variables are encoded through one-hot encoding, except for variables with twenty or more categories, which were removed (this applies to the KDD census dataset). The binarized datasets are included in our code repository (if the license permits redistribution). The Diabetes and Law-school dataset are left out of the evaluation because the best tree of depth three is (almost) identical to the trivial solution (assign the majority label to all instances). See Appendix 3.C for more details.

The algorithm receives the whole dataset as input and is asked to find a tree with an unfairness limit of $\delta = 1\%$. In a later experiment, we also examine the impact of this parameter on the runtime. The values shown are the average of five runs and a time limit of one hour is set for every run. All experiments are run on a 2.6Ghz Intel i7 CPU with 8GB RAM using only one thread.

The FairOCT model To our knowledge, the only methods available for finding fair and optimal decision trees are MIP models (Verwer and Zhang, 2017; Aghaei et al., 2019; Jo et al., 2023). Verwer and Zhang (2017) limit their model to solving problems with less than 1000 instances. In our initial comparison of Aghaei et al. (2019) and Jo et al. (2023), the latter outperformed the first by a large margin. Therefore, we compare our method with their FairOCT model. They model the decision tree as a flow graph where all instances must flow from the source (connected to the root node of the tree) to one of the sink nodes (one for each class), to which all nodes are connected. In our experiments, the FairOCT model is solved with Gurobi 9.0 using the default parameters.

Runtime comparison Table 3.1 shows the runtime comparison between FairOCT and our method DPF. DPF can find optimal trees for $d = 2$ within one second for all datasets, about four to five orders of magnitude faster than FairOCT. FairOCT is able to find $d = 2$ trees only for the smallest datasets within one hour. It struggles with an increasing number of data instances because every data instance is represented by a binary variable. Also for $d = 3$, DPF can find optimal trees within one second for several datasets and all problems are solved within a minute. For $d = 4$, the exponential nature of the problem becomes clearly visible. Datasets with both large number of features and instances are no longer solvable within one hour. However, DPF still succeeds in finding the best tree for a number of datasets, among which Adult, with over 45000 instances.

				FairOCT	DPF		
Dataset	a	$\lvert\mathcal{D}\rvert$	$\lvert\mathcal{F}\rvert$	$d=2$	$d=2$	$d=3$	$d=4$
Adult	Gender	45222	17	$>$ 1h	<1	<1	1012
Bank Marketing	Married	45211	46	$>$ 1h	<1	2	$>$ 1h
Communities & Crime	Race	1994	97	$>$ 1h	<1	4	1261
COMPAS recid.	Race	6172	9	$>$ 1h	<1	<1	9
COMPAS viol. recid.	Race	4020	9	1594	<1	<1	<1
Dutch census	Gender	60420	58	$>$ 1h	<1	5	$>$ 1h
German credit	Gender	1000	69	$>$ 1h	<1	2	741
KDD census income	Race	284556	117	-	<1	32	$>$ 1h
OULAD	Gender	21562	45	$>$ 1h	<1	3	$>$ 1h
Ricci	Race	118	4	<1	<1	<1	<1
Student-Mathematics	Gender	395	55	284	<1	<1	24
Student-Portuguese	Gender	649	55	866	<1	<1	32

Table 3.1: Runtime comparison of DPF and FairOCT. Runtime is in seconds. FairOCT resulted in an out of memory error for KDD census income, so no runtime can be reported.

The impact of our novel pruning method and the merge improvements Table 3.2 shows the runtime results when DPF is run 1) without our novel pruning method (not applying the prune and nondom methods of Eqs. (3.7) and (3.17); 2) without our improvements to the merge method; 3) without our improvement to the nondom method, and 4) to generate a full Pareto front.

For several datasets, our pruning method decreases the runtime by one or more orders of magnitude. This is specifically the case for depth four, where the subproblem dependency is more significant than for depth three. Our method can find optimal trees for depth four for eight out of twelve datasets within one hour, but without the pruning method, this is possible for only four datasets.

Our improvements to the merge function in some cases significantly reduce the runtime. This is specifically the case for datasets that have a high initial bias (the bias of an optimal tree if fairness would not be considered): Adult, COMPAS recid. and Dutch census have an initial bias of 18%, 16%, and 14% respectively, and for these datasets the merge improvements reduce the runtime by an order of magnitude. The other datasets in Table 3.2 have an initial bias of 1-6% and the merge improvements result in only a small reduction of the runtime. The improvements to the nondom function are also essential for the performance of our pruning mechanism.

Generating the full Pareto-front Because of the runtime improvements of DPF, it is also able to find a *full*, fair and optimal decision tree Pareto front. In contrast, Valdivia et al. (2021) find a partial and non-optimal Pareto front and the FairOCT model is used to find an optimal, but still partial, Pareto front. Table 3.2 shows the runtime performance of DPF when tasked to find the full Pareto front. In comparison to FairOCT, DPF is several orders of magnitude faster. For COMPAS recid., for example, DPF found the Pareto front for depth two consisting of 29 solutions in 0.12 ± 0.04 seconds. The FairOCT method generates the partial Pareto front by repeatedly solving the problem with a different maximum bias, resulting in

	$d = 3$						$d = 4$				
Dataset	¬P	¬M	¬D	Pr	Df	Dataset	¬P	¬M	¬D	Pr	Df
Bank	5	3	319	4	2	Adult	> 1h	> 1h	> 1h	> 1h	1012
Com.&Cr.	13	5	19	7	4	Com.&Cr.	> 1h	1329	2473	1939	1261
Dutch	> 1h	101	1814	510	5	COMP. r.	> 1h	394	1658	1659	9
German	8	2	22	2	2	German	> 1h	770	1971	738	741
KDD	144	35	1254	50	32	Stud. Math	144	24	29	412	24
OULAD	5	4	1448	3	3	Stud. Port.	1834	32	38	259	32

Table 3.2: DPF runtimes without the pruning and dominance checks (¬P), without the merge improvements (¬M), without the nondom improvements (¬D), for finding the full Pareto-front (Pr), and for comparison the default runtime (Df). Datasets for which DPF has a default runtime below one second or over one hour, are left out.

a runtime several times higher than the values reported in Table 3.1.

See Figure 3.1 for a number of full Pareto fronts generated by DPF for depth 2-4. These plots confirm the trade-off between accuracy and fairness as reported in previous works.

Method evaluation Figure 3.2 shows the impact of the number of features, the number of dataset instances, the minimum leaf node size and the maximum allowed bias in the fairness constraint on the runtime of DPF for three datasets when searching for trees of depth three. It can be observed from these experiments that the number of features is the most important factor, resulting in an exponential increase, which is in line with previous studies (Lin et al., 2020; Demirović et al., 2022). One might expect that the method would scale linearly with increasing dataset size, but the experiments show almost no increase in runtime after a certain size. The reason for this is that the runtime of the method is mostly dependent on the number of unique partial solutions that need to be merged by Eq. (3.9). This number of unique solutions is strongly dependent on the number of features and is only weakly related to the number of instances.

DPF can easily be changed to limit the minimum leaf node size, by returning an empty set in Eq. (3.15) when the number of instances does not exceed the required leaf node size. Increasing the minimum leaf node size decreases the amount of unique partial solutions, and consequently we see a significant decrease in the runtime when

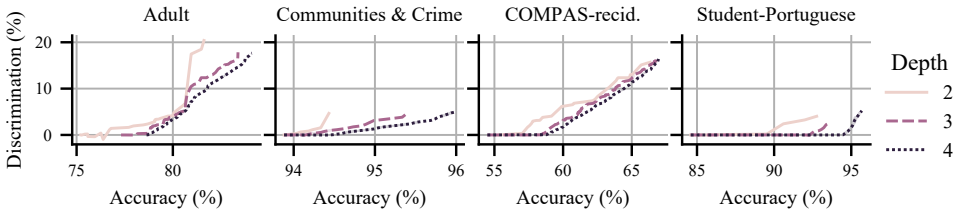


Figure 3.1: The full Pareto front of training accuracy and discrimination for four datasets for trees of depth 2, 3 and 4, generated by DPF.

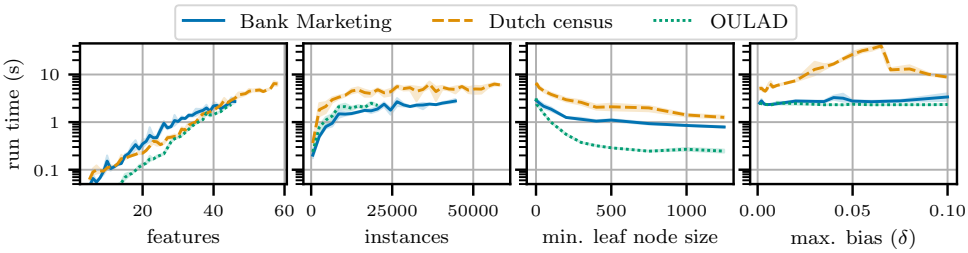


Figure 3.2: The impact of the number of features, instances, minimum leaf node size and maximum allowed bias in the fairness constraint, δ, on the runtime of DPF with $d = 3$. The shaded area shows the values within one standard deviation. Note the logarithmic scale of the y-axis.

the minimum leaf node size is increased. Adding such a minimum leaf node size often does not have a significant impact on the accuracy score.

Relaxing the fairness constraint to values larger than $\delta = 1\%$ causes the runtime to increase for the Dutch census dataset. This can be explained because a more relaxed constraint means a larger search space and less strong fairness bounds. However, when the constraint is even more relaxed, the problem is closer to a normal optimal tree search, and the accuracy bounds become stronger, again resulting in a shorter runtime. For the Bank marketing and OULAD datasets, the default discrimination when no fairness constraint is in place, is lower: 1.2% and 3.0%, versus Dutch census: 14.1%. This explains why these datasets show almost no difference in runtime when changing the value δ.

3.6. Conclusion

We show how dynamic programming can be used to find fair and optimal decision trees, even when considering a global fairness constraint that introduces subproblem dependency. We solve the subproblem dependency by computing upper and lower bounds on the final fairness value and thus enable comparison of partial solutions. The results show that our method DPF can find fair optimal trees several orders of magnitude faster than the state of the art. As a result, DPF succeeds in finding fair and optimal decision trees even for large datasets that were previously beyond reach. Moreover, we present the first specialized algorithm for finding fair optimal decision trees that can also find the full Pareto front in terms of accuracy and fairness.

An extension of DPF would be to support other group-based notions of fairness, such as equality of opportunity. To better deal with unbalanced datasets, another extension of the presented research would be to replace the accuracy objective with an objective that can cope better with unbalanced datasets, for example, with the biobjective method presented by Demirović and Stuckey (2021), or by using balanced accuracy. Finally, based on the evaluation results, new heuristic methods could be investigated, for example, by considering not all possibly-fair partial solutions, but only a representative subset.

Appendices for Chapter 3

3.A. Complexity Analysis

The runtime of DPF as presented in Algorithm 5 depends on many factors and specifically on the number of feasible solutions and the amount of solutions that can be pruned early in the search. We here provide a worst case bound that assumes that all trees are feasible and no solution can be pruned. An upper bound of the number of unique complete trees of depth d is given by $n_t = O(|\mathcal{F}|^{2^d-1}2^{2^d})$: i.e., the product of the number of possible branching decision assignments and leaf node label assignments. In the worst case scenario, no tree would be pruned and the nondom operation would compare every tree with every other tree: n_t^2 comparisons. A second term in the runtime complexity is from the recursive tree search calls. For $d > 0$, DPF has $2|\mathcal{F}|$ recursive calls, resulting in a total of $2^d|\mathcal{F}|^d$ calls to DPF. Each of these calls traverses the dataset once. Therefore a worst case runtime complexity for DPF is: $O(n_t^2 + 2^d|\mathcal{F}|^d|\mathcal{D}|)$. In practice, however, the runtime is much smaller because of pruning.

3.B. Non-Complete Trees

For the sake of brevity, the main text of this chapter only shows how to search for complete trees, that is, trees of depth d with $2^d - 1$ branch nodes and 2^d leaf nodes. Our method also allows to search for smaller trees, as was done similarly by Demirović et al. (2022).

The DP formulation in Eq. (3.16) can be extended as follows to also allow for incomplete/sparse trees. Here n signifies the number of nodes in the tree.

$$\begin{aligned}&\text{nondom}\,\big(\text{prune}\,\big(\cup_{f\in\mathcal{F},i\in[0,n-1]}\,\text{merge}\,(\\&\quad T_F(\mathcal{D}_{\bar{f}}, d-1, n-i-1, \quad U(\mathcal{D}_f, [\underline{I}_R, \bar{I}_R], d-1, i)),\\&\quad T_F(\mathcal{D}_f, d-1, i, \quad U(\mathcal{D}_{\bar{f}}, [\underline{I}_R, \bar{I}_R], d-1, n-i-1))))\big)\end{aligned} \tag{3.21}$$

With this change in place, the solver can search for incomplete trees. This also allows to add a parameter α to prevent overfitting. The misclassification score now becomes $M + \alpha n$. The parameter α describes how much the misclassification score should at least decrease in order to justify adding another node to the tree. The addition of this parameter to the algorithm is trivial.

3.C. Dataset Details

Table 3.3 shows detailed information about every dataset considered in this study. The references for the original datasets can be found here (Dutch Central Bureau for Statistics, 2001a; Cortez and Silva, 2008; Moro et al., 2014; Strack et al., 2014; Angwin et al., 2016a; Dua and Graff, 2017; Kuzilek et al., 2017).

Name	$\lvert\mathcal{D}\rvert$	$\lvert\mathcal{F}\rvert$	protected feature	y=1 $a=1$	y=1 $a=0$	y=0 $a=1$	y=0 $a=0$	Acc. d=3	Disc. d=3
Adult	45222	17	Gender	9539	1669	20988	13026	83.4	-17.8
Bank	45211	46	Married	2755	2534	24459	15463	90.0	1.2
Com.&Cr.	1994	97	Race	1017	855	7	115	95.4	-4.5
COMP. r.	6172	9	Race	1281	2082	822	1987	66.7	-16.4
COMP. v.r.	4020	9	Race	1285	2083	174	478	84.1	-1.6
Dutch	60420	58	Gender	18860	9903	11287	20370	81.4	-14.1
German	1000	69	Gender	499	201	191	109	75.3	-5.5
KDD	284556	117	Race	15926	1475	223155	44000	94.4	-1.1
OULAD	21562	45	Gender	7727	6928	3841	3066	69.1	-3.0
Ricci	118	4	Race	41	15	27	35	100	-30.3
Stud. Math	395	55	Gender	132	133	55	75	93.4	-6.3
Stud. Port.	649	55	Gender	216	333	50	50	93.7	3.4

Table 3.3: Datasets used for evaluation. Preprocessed as described in Le Quy et al., 2022. Training accuracy (%) and discrimination results (%) are shown for the best tree of depth three that does not consider fairness when training with the full dataset. The sign of the discrimination score tells which of the two groups is discriminated against.

3.D. Test Evaluation

Experiment setup The main text evaluation focuses on analyzing the runtime of our DPF method. This section further analyzes the out-of-sample performance of DPF and compares it with two heuristics. The pre-processing (massaging) approach presented by Kamiran and Calders (2009), and a post-processing approaches proposed by Kamiran et al. (2010). We will call the pre-relabelling method KamPre and the post-relabelling method KamPost, after their first author (Kamiran).

KamPre pre-processes the training data by relabeling a number of instances in the training data such that the training data no longer is biased. Like them, we use a Naive Bayes classifier to decide which labels to change. Unlike them, we use the newly labeled data to train a standard decision tree with CART.

KamPost differs from CART by using a different splitting criterion and by its post-relabelling of the leaf nodes. KamPost uses a splitting criterion that is a mix of information gain in the class label and information gain in the sensitive attribute (IGC+IGS). After generating the tree, it heuristically changes the label of some leaves in such a way that discrimination is minimized with the least loss of accuracy.

We do not compare to the optimal MIP method FairOCT because its optimal solutions would either be the same as those generated by DPF, or, if multiple optimal solutions exists, any difference could only be attributed to a random selection of one out of several optimal models.

We first evaluate the out-of-sample performance by running DPF for $d = 2, 3, 4$ and KamPost for $d = 3, 4$ for different maximum allowed bias $\delta = 1\%, 5\%, 100\%$ on three datasets that are commonly evaluated in the literature. Note that KamPost with $\delta = 100\%$ is the same as plain CART. We run each case 10 times on random stratified train-test splits of 75% vs 25%. We here do not compare to KamPre

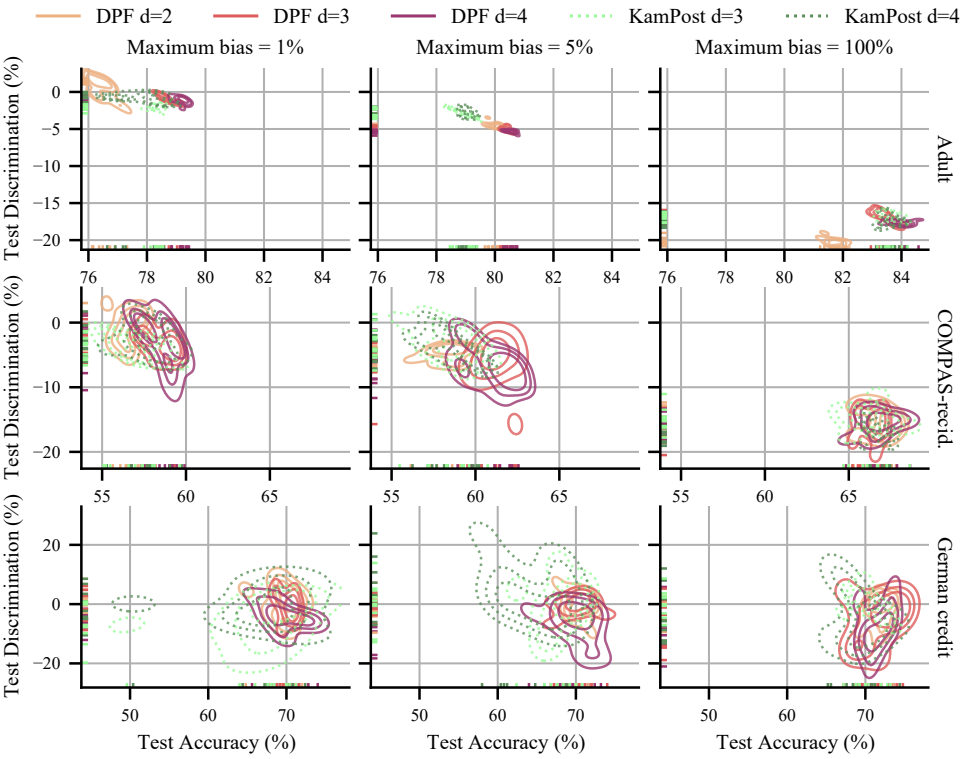


Figure 3.3: Out-of-sample accuracy and discrimination for three datasets, for maximum bias $\delta = 1\%, 5\%, 100\%$. The figure shows the distribution of 10 runs.

because it does not take a maximum allowed bias as an input factor. Figure 3.3 shows the resulting distribution of test accuracy and test discrimination. The sign of the discrimination score tells which of the two groups is discriminated against.

Evaluation From the results in Figure 3.3 it can be observed that the optimal decision trees generated by DPF in general have better out-of-sample accuracy than KamPost for the same or smaller depth. In several cases DPF $d = 2$ even outperforms KamPost, even though it uses four times less nodes. However, in general the variance in test accuracy is high.

The variance in test discrimination is also high, and both methods often exceed the imposed discrimination limit in the test evaluation. This problem is less visible with the Adult dataset, probably because of its larger number of data instances.

For both the COMPAS-recid. and German credit dataset almost maximum accuracy can already be obtained with a decision tree of depth two, so the addition of extra nodes does not help much. For the Adult dataset, however, deeper trees can provide better accuracy.

Tuned for number of nodes In our next analysis, we prevent overfitting during training for all three methods by using 10 random validation splits of 25% of the

	DPF		KamPre		KamPost	
Dataset	Accuracy	Disc.	Accuracy	Disc.	Accuracy	Disc.
Adult	80.4 ± 0.3	-5.1 ± 0.4	75.2 ± 0.0	0.0 ± 0.0	**78.0** ± 1.5	-1.9 ± 1.1
Bank	**89.7** ± 0.1	1.7 ± 0.6	89.1 ± 0.3	0.9 ± 0.6	89.0 ± 0.4	0.5 ± 0.3
Com.&Cr.	**94.6** ± 0.3	-3.2 ± 1.0	93.0 ± 1.1	-3.3 ± 2.6	93.9 ± 0.3	-1.3 ± 2.3
COMP. r.	**59.1** ± 2.4	-4.6 ± 3.1	61.4 ± 1.8	-9.8 ± 2.8	55.6 ± 1.9	-1.1 ± 2.1
COMP. v.r.	83.7 ± 0.2	-0.3 ± 0.5	82.2 ± 1.1	-4.0 ± 2.1	83.8 ± 0.1	-0.3 ± 0.8
Dutch	**77.4** ± 0.3	-5.0 ± 1.0	76.0 ± 0.2	-9.9 ± 0.4	68.2 ± 10.9	-0.6 ± 0.6
German	70.3 ± 1.7	-2.1 ± 3.0	69.8 ± 1.3	-0.7 ± 3.9	70.3 ± 1.0	-0.9 ± 3.4
KDD	**94.3** ± 0.0	-1.0 ± 0.1	93.9 ± 0.0	0.0 ± 0.0	93.9 ± 0.0	0.0 ± 0.1
OULAD	**68.7** ± 0.3	-2.1 ± 1.0	68.2 ± 0.6	-1.8 ± 1.1	68.0 ± 0.1	0.1 ± 0.2
Ricci	66.0 ± 4.4	-13.4 ± 12.3	100.0 ± 0.0	-28.1 ± 0.0	**53.3** ± 0.0	0.0 ± 0.0
Stud. Math	**85.5** ± 6.6	-2.8 ± 8.3	89.7 ± 2.8	-5.5 ± 3.2	73.9 ± 11.8	-2.5 ± 4.2
Stud. Port.	90.5 ± 2.8	0.8 ± 5.0	91.2 ± 4.2	6.1 ± 4.9	89.4 ± 4.2	2.3 ± 3.1

Table 3.4: Out-of-sample average accuracy and discrimination ± the standard deviation (%) for solutions with a maximum training bias of $\delta = 5\%$ and a maximum depth of $d = 3$. Whenever the 5% discrimination threshold is exceeded on average, the result is marked in red. Best performing accuracy score per dataset is marked bold, if significantly better than other methods that also stay within the 5% discrimination limit (p-value $< 5\%$).

training data to find what number of branching nodes is best. The tree size with the best average accuracy in the validation set, while on average respecting the discrimination constraint is selected as best. The full training dataset is then used to generate a tree of that size. Table 3.4 shows the results when all three algorithms are used to find trees of a maximum bias of 5%.

Discussion DPF searches for optimal decision trees, which means it will always find the tree with maximum accuracy for the training dataset, and thus always outperform heuristics on performance in the training dataset. The results in Table 3.4 show that when DPF is tuned for selecting the right number of nodes, this on average also generalizes to better performance than KamPre and KamPost in the test set.

We compare the results of the methods that achieve (on average) a test discrimination lower than 5%, and among those select the method with highest test accuracy. There are seven datasets for which DPF is significantly better than KamPre and KamPost ($p < 5\%$), with differences in accuracy even as large as 11.6% (Student-Mathemtacis) or 9.2% (Dutch census). KamPost only scores best for Ricci and Adult. Ricci is the smallest of all datasets with only 118 instances and 4 features, but KamPost's result is only slightly better than random. KamPost also performs best for Adult, with DPF exceeding the limit by 0.1%. For three datasets, no method is significantly better than the others.

The results also confirm the findings from Figure 3.3 that the variance in the discrimination value is often high, specifically for the small datasets. This means that for those instances it is difficult to generalize and overfitting in terms of discrimination is still happening. It is an open question how this can be reduced.

4

A General DP Approach: Theory and Applications

Preface This chapter develops the method of Chapter 3 into a general framework. It discovers necessary and sufficient conditions for the use of dynamic programming in ODT learning. The framework makes it easier to model new problems, and we apply it to and empirically compare it with baseline methods for five different machine learning tasks.

Abstract Global optimization of decision trees has shown to be promising in terms of accuracy, size, and consequently human comprehensibility. However, many of the methods used rely on general-purpose solvers for which scalability remains an issue. Dynamic programming methods have been shown to scale much better because they exploit the tree structure by solving subtrees as independent subproblems. However, this only works when an objective can be optimized separately for subtrees. We explore this relationship in detail and show the necessary and sufficient conditions for such separability and generalize previous dynamic programming approaches into a framework that can optimize any combination of separable objectives and constraints. Experiments on five application domains show the general applicability of this framework, while outperforming the scalability of general-purpose solvers by a large margin.

4.1. Introduction

Many high-stakes domains, such as medical diagnosis, parole decisions, housing appointments, and hiring procedures, require human-comprehensible machine learning (ML) models, such as decision trees, to ensure transparency, safety, and reliability (Rudin, 2019). However, these application domains wildly differ from one another,

Parts of this chapter have been published in Van der Linden, De Weerdt, and Demirović (2023), "Necessary and Sufficient Conditions for Optimal Decision Trees using Dynamic Programming", in *Advances in NeurIPS-23*. In the paper, most of the experiments were originally part of the appendix, but for this chapter those experiments have become part of the main experiment section.

and therefore the objective and constraints also differ, leading to a variety of tasks: e.g., cost-sensitive classification for medical diagnosis (Turney, 1995; Freitas et al., 2007); prescriptive policy generation for revenue maximization (Biggs et al., 2021); or ensuring fairness constraints are met (Aghaei et al., 2019). For broad-scale application of optimal decision trees, we need methods that can *generalize* to incorporate objectives and constraints such as the ones from these application domains, while remaining *scalable* to real-world problem sizes.

Dynamic programming (DP) approaches (Aglin et al., 2020a; Lin et al., 2020; Demirović et al., 2022) show scalability that is orders of magnitude better than alternatives, such as MIP, SAT, and CP, by directly exploiting the tree structure. Each problem is solved by a recursive step that involves two subproblems, each just half the size of the original problem. Additional techniques such as bounding and caching are used to further enhance performance.

However, unlike general-purpose solvers such as MIP, DP cannot easily accommodate the variety objectives and constraints mentioned before; it requires the optimization task to be solvable *separately* for each subtree to remain efficient.

Therefore, the main research question considered here is to what extent can this separability property be generalized? In answering this question we provide a general framework called *STreeD* (Separable Trees with Dynamic programming). We push the limits of DP for optimal decision trees by providing conditions for separability that are both necessary and sufficient. These conditions are less strict and extend to a larger class of optimization tasks than those of state-of-the-art DP frameworks (Nijssen and Fromont, 2010; Lin et al., 2020). We also show that any combination of separable optimization tasks is also separable. We thus attain generalizability similar to general-purpose solvers, while preserving the scalability of DP methods.

In our experiments, we demonstrate the flexibility of STreeD on a variety of optimization tasks, including cost-sensitive classification, prescriptive policy generation, nonlinear classification metrics, and group fairness.

As is commonly done, we assume that features are binarized beforehand: STreeD returns optimal *binary* decision trees under the assumption of a given binarization.

In summary, our main contributions are (i) a generalized DP framework (STreeD) for optimal decision trees that can optimize any *separable* objective or constraint; (ii) a proof for necessary and sufficient conditions for separability; and (iii) extensive experiments on five application domains that show the flexibility of STreeD, while performing on par or better than the state of the art.

The following sections introduce related work and preliminaries. We then further define separability, show conditions for separability, and provide a framework that can find optimal decision trees for any separable optimization task. Finally, we provide five diverse example applications and test the performance of the new framework on these applications versus the state of the art.

4.2. Related Work

In this related work, we review how decision tree learning is extended and adapted for applying a variety of constraints and objectives. In Chapter 2, we provide related

work for (optimal) decision tree learning in general.

Heuristics and constraints Traditional tree induction heuristics, such as CART (Breiman et al., 1984) and C4.5 (Quinlan, 1993b), often employ a top-down compilation, splitting the tree based on a local information gain or entropy metric. This technique is ill-suited for enforcing global constraints, since each split only considers local information. Additionally, the splitting criterion typically does not optimize the real objective or constraint directly, but a proxy (e.g., information gain is a proxy of accuracy). Therefore, for greedy heuristics, for each new objective, new splitting criteria need to be devised, and often this splitting criterion only indirectly relates to the real objective.

Objectives and constraints Nanfack et al. (2022) categorize the constraints used in decision tree learning into structure, attribute, and instance constraints. Structure constraints limit the structure of the tree, such as its depth or size. Attribute constraints impose limits based on the attribute values. Examples are monotonicity constraints (Hu et al., 2011), cost-sensitive classification (Lomax and Vadera, 2013) and group fairness constraints (Jo et al., 2023). Instance constraints operate on pairs of instances, such as robustness constraints (Vos and Verwer, 2022), individual fairness constraints (Aghaei et al., 2019), and must- or cannot-link constraints in clustering (Struyf and Džeroski, 2007). Among these, structure constraints are most widely included in decision tree learning methods.

Many of these constraints or objectives can be added to MIP models. Therefore, the literature shows a multitude of MIP models for a variety of applications, such as fairness (Aghaei et al., 2019), algorithm selection (Vilas Boas et al., 2021), and robustness (Vos and Verwer, 2022). However, scalability remains an issue.

Nijssen and Fromont (2010) show how the DP-based DL8 algorithm (Nijssen and Fromont, 2007) can be generalized for a variety of objectives, provided that the objectives are *additive*, i.e., the value of an objective in a tree node is equal to the sum of the value in its children. Constraints are required to be *anti-monotonic*, i.e., a constraint should always be violated for a tree whenever it is violated for any of its subtrees. They consider constraints and objectives such as minimum support in a leaf node, C4.5's estimated error rate, Bayesian probability estimation, cost-sensitive classification, and privacy preservation.

Since then, new DP methods have improved on the runtime performance of DL8. This, however, reduced the generalizability of those methods. DL8.5 (Aglin et al., 2020a,b), for example, can optimize any additive objective but does not support constraints. MurTree (Demirović et al., 2022) provides new algorithmic techniques that result in better scalability but only considers accuracy. GOSDT (Lin et al., 2020) can optimize additive objectives that are linear in the number of false positives and negatives but also does not support constraints. When the objective is nonlinear, they no longer use DP. Demirović and Stuckey (2021) show how DP can be used to optimize even nonlinear functions, provided the objective is monotonically increasing in terms of the false positives and negatives, but they also do not consider constraints.

Summary Decision tree learning is often approached through top-down induction heuristics, which depend on developing complex custom splitting criteria for each

new learning task and may perform suboptimally in terms of classification accuracy. Optimal methods often outperform heuristics in accuracy but typically lack scalability. Optimal DP approaches have better scalability but lack generalizability to other objectives and constraints. In the next sections, we address this by pushing the limits of what can be solved by a general yet efficient DP framework for optimal decision trees.

4.3. Preliminaries

This section introduces notation, defines the problem, and explains how dynamic programming solutions for optimal decision trees work.

Notation and problem definition Let $\mathcal{F}$ be a set of features and let $\mathcal{K}$ be a set of labels that are used to describe an instance. Let $\mathcal{D}$ be a dataset consisting of instances (x, k), with $x \in \{0,1\}^{|\mathcal{F}|}$ the feature vector and $k \in \mathcal{K}$ the label of an instance. Because the features are binary, we introduce the notation f and $\bar{f}$ for every feature $f \in \mathcal{F}$ to denote whether an instance satisfies feature f or not. In the same way, let $\mathcal{D}_f$ describe the set of instances in $\mathcal{D}$ that satisfy feature f, and $\mathcal{D}_{\bar{f}}$ the set of instances that does not.

Let $\tau = (B, L, b, l)$ be a binary tree with B the internal branching nodes, L the leaf nodes, $b : B \to \mathcal{F}$ the assignment of features to branching nodes and $l : L \to \mathcal{K}$ the assignment of labels to leaf nodes. The left and right child nodes of a branching node $u \in B$ are given by u_L and u_R, respectively. Instances that satisfy a feature branching test are sent to the right subtree, and the rest to the left.

Then, for example, when minimizing misclassification score (the number of misclassified instances), the cost C of a decision tree can be computed by the recursive formulation:

$$\mathrm{C}(\mathcal{D}, u) = \begin{cases} \sum_{(x,k)\in\mathcal{D}} \mathbb{1}(k \neq l(u)) & \text{if } u \in L \\ \mathrm{C}(\mathcal{D}_{\overline{b(u)}}, u_L) \ + \mathrm{C}(\mathcal{D}_{b(u)}, u_R) & \text{if } u \in B \end{cases} \tag{4.1}$$

The task is to find an optimal decision tree τ that minimizes the objective value over the training data given a maximum tree depth d. We now explain how this can be optimized with DP, and in the next section, we generalize this to any separable decision-tree optimization task.

Dynamic programming DP simplifies a complex problem by reducing it to smaller repeated subproblems for which solutions are cached. For example, minimizing misclassification score with a binary class can be solved with DP as follows (adapted from (Demirović et al., 2022)):

$$T(\mathcal{D}, d) = \begin{cases} \min\{|\mathcal{D}^+|, |\mathcal{D}^-|\} & d = 0 \\ \min_{f\in\mathcal{F}} \{\ T(\mathcal{D}_f, d-1) \ + \ T(\mathcal{D}_{\bar{f}}, d-1)\} & d > 0 \end{cases} \tag{4.2}$$

This equation computes the minimum possible misclassification score for a dataset $\mathcal{D}$ and a given maximum tree depth d. Recursively, as long as $d > 0$, a feature f is selected for branching such that the sum of the misclassification score of the left and

right subtree is minimal. In the leaf node (when $d = 0$), the label of the majority class is selected: either the positive ($\mathcal{D}^+$) or negative ($\mathcal{D}^-$) label. This approach is further optimized by using caching and bounds.

As is common with DP notation, Eq. (4.2) returns the solution value (or cost) and not the solution (the tree). In the remainder of the text, we refer to solution values and solutions interchangeably, with one implying the other and vice versa.

While Eq. (4.2) yields a single optimal solution, Demirović and Stuckey (2021) investigated the use of DP for nonlinear bi-objective optimization by searching for a Pareto front of optimal solutions: i.e., the set of solutions that are not Pareto dominated ($\succ$) by any other solution:

$$\text{nondom}(\Theta) = \{v \in \Theta \mid \neg\exists\ v' \in \Theta\ (v' \succ v)\} \tag{4.3}$$

Consequently, every subtree search also no longer returns just one solution but a Pareto front. For this, they introduce a merge function that combines every solution from one subtree with every solution from the other subtree, resulting in a new Pareto front.

4.4. Framework for Separable Objectives

In this section, we generalize previous DP methods for optimal decision trees to a new general framework that can solve any separable optimization task. First, we define the problem. Second, we formalize what is meant by separability in the context of learning decision trees and then we prove necessary and sufficient conditions for optimization tasks to be separable. Third, we present the generalized framework. Finally, we show separable optimization task formulations for four example domains.

4.4.1. Problem Definition and Notation

We now generalize the previously introduced problem of minimizing misclassification score to any decision tree optimization task by using common DP notation. Given a state space $\mathcal{S}$ and a solution space $\mathcal{V}$, we define:

Definition 4.4.1 (Optimization task)**.** An *optimization task* is described by six components:

1. a cost function $g: \mathcal{S} \times (\mathcal{F} \cup \mathcal{K}) \to \mathcal{V}$ that returns the cost of action $a \in \mathcal{F} \cup \mathcal{K}$ in state $s \in \mathcal{S}$, where the action is either assigning label $\hat{k} \in \mathcal{K}$ or branching on feature $f \in \mathcal{F}$;
2. a transition function $t: \mathcal{S} \times \mathcal{F} \times \{0, 1\} \to \mathcal{S}$ that provides the next state after branching left or right on feature f, denoted by f or $\bar{f} \in \mathcal{F} \times \{0, 1\}$;
3. a comparison operator $\succ: \mathcal{V} \times \mathcal{V} \to \{0, 1\}$ that determines Pareto dominance;
4. a combining operator $\oplus: \mathcal{V} \times \mathcal{V} \to \mathcal{V}$ that combines solution values to one value;
5. a constraint $c: \mathcal{V} \times \mathcal{S} \to \{0, 1\}$ that determines feasibility of a given solution v and state s;
6. and an initial state $s_0 \in \mathcal{S}$.

For all optimization tasks in this chapter, the state s can be described by the dataset $\mathcal{D}$ and the branching decisions F in parent nodes. The transition function is given by $t(\langle \mathcal{D}, F \rangle, f) = \langle \mathcal{D}_f, F \cup \{f\} \rangle$, and the initial state s_0 is $\langle \mathcal{D}, \emptyset \rangle$. But our framework extends beyond this as well.

When minimizing misclassifications, the cost function is defined as $g(\langle \mathcal{D}, F \rangle, \hat{k}) = |\{(x, k) \in \mathcal{D} \mid k \neq \hat{k}\}|$; the comparison operator $\succ$ is $<$; the combining operator $\oplus$ is addition and the constraint c returns all solutions as feasible. Similarly, more complex objectives, such as F1-score and group fairness, which do not fit the conditions of DL8 (Nijssen and Fromont, 2010), can be defined by using these building blocks. We show four examples in Section 4.4.4.

The final cost of a tree $\tau = (B, L, b, l)$ with root node r is now given by calling $\mathrm{C}(s_0, r)$ on the following recursive function, which generalizes Eq. (4.1):

$$\mathrm{C}(s, u) = \begin{cases} g(s, l(u)) & \text{if } u \in L \\ \mathrm{C}(t(s, \overline{b(u)}), u_L) \ \oplus \ \mathrm{C}(t(s, b(u)), u_R) \ \oplus \ g(s, b(u)) & \text{if } u \in B \end{cases} \tag{4.4}$$

In leaf nodes, the cost of a tree is given by cost function g. In branching nodes, the combining operator $\oplus$ combines the cost of the left and right subtree and the branching costs.

Given an optimization task $o = \langle g, t, \succ, \oplus, c, s_0 \rangle$ and a maximum tree depth d, the aim is to find a tree (or a Pareto front of trees) that satisfies the constraint c and optimizes (according to the comparison operator $\succ$) the cost C.

With this definition of an optimization task, we can generalize several of the concepts introduced in the preliminaries. Let Θ describe a set of possible solutions and let

$$\mathrm{feas}(\Theta, s) = \{v \in \Theta \mid c(v, s) = 1\} \tag{4.5}$$

be the subset of all feasible solutions in Θ for state s. The optimal set of solutions is given by

$$\mathrm{opt}\,(\Theta, s) = \mathrm{nondom}(\mathrm{feas}(\Theta, s)), \tag{4.6}$$

the Pareto front of feasible solutions in Θ, with nondom depending on the comparison operator $\succ$. Furthermore, we generalize the definition for merge from (Demirović and Stuckey, 2021):

$$\mathrm{merge}(\Theta_1, \Theta_2, s, f) = \{v_1 \oplus v_2 \oplus g(s, f) \mid v_1 \in \Theta_1, v_2 \in \Theta_2\} \tag{4.7}$$

This returns all combinations of solutions from sets Θ_1 and Θ_2 by using the combining operator $\oplus$.

4.4.2. Separability

An optimization task is separable if optimal solutions to subtrees can be computed independently of other subtrees. For example, in the tree in Fig. 4.1, the optimal label L3 should be independent of the branching decision F2 and labels L1 and L2. Label L2 should be independent of L1 and L3.

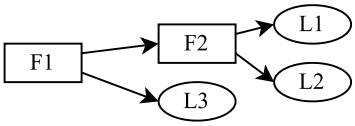


Figure 4.1: An example tree of depth two with five decision variables: two branching decisions F1 and F2, and three leaf node label assignments: L1, L2, and L3.

Definition 4.4.2 (Separable)**.** An optimization task is *separable* if and only if the optimal solution to any subtree can be determined independently of any of the decision variables that are not part of that subtree or the parent nodes' branching decisions.

The general idea to prove that an optimization task satisfies Def. 4.4.2 is to use complete induction over the maximum tree depth d. For the base step ($d = 0$) it is sufficient to require the cost and transition functions to be *Markovian*: their output should depend only on the current state and the chosen decision. For the induction step ($d > 0$), we need to show that the optimization task satisfies the *principle of optimality* (Bellman, 1957): that optimal solutions for trees of depth d can be constructed from only the optimal solutions to its subtrees and the new decision (see Def. 4.A.1). In summary:

Proposition 4.4.3. *An optimization task* $o = \langle g, t, \succ, \oplus, c, s_0 \rangle$ *is* separable *if and only if its cost function* g *and transition function* t *are* Markovian *and the task* o *satisfies the* principle of optimality*.*

A full proof for this and all subsequent propositions and theorems can be found in Appendix 4.A.

To satisfy the *principle of optimality*, the combining operator $\oplus$ must *preserve order* over $\succ$, and the constraint c must be *anti-monotonic*. These two notions are explained next.

We introduce the new notion *order preservation* that guarantees that any combination of optimal solutions using the combining operator $\oplus$ always dominates a combination with at least one suboptimal solution. Many objectives, such as costs, are *additive* with $<$ as the $\succ$ operator. Addition preserves order over $<$. However, *addivity* is not a necessary condition. We present *order preservation* instead as a necessary condition (see Appendix 4.A.2).

Definition 4.4.4 (Order preservation)**.** A combining operator $\oplus$ *preserves order* over a given comparator $\succ$, if for any given state s, feature f and solution sets Θ_1 and Θ_2, with $s_1 = t(s, f)$, $s_2 = t(s, \bar{f})$, $v_1 \in \text{opt}(\Theta_1, s_1), v_1' \in \Theta_1$ and $v_2 \in \text{opt}(\Theta_2, s_2)$, with $v_1 \succ v_1'$, then $v_1 \oplus v_2 \succ v_1' \oplus v_2$ (and $v_2 \oplus v_1 \succ v_2 \oplus v_1'$, if $\oplus$ is not commutative).

To guarantee that applying feas as defined in Eq. (4.5) does not prune any partial solution that is part of the final optimal solution, constraint c must be *anti-monotonic* (Nijssen and Fromont, 2010): if a constraint is violated in a tree, it is also violated in any tree of which this tree is a subtree. However, in order to speak of a *necessary* condition, we redefine anti-monotonicity to require that any solution

which is constructed from at least one subsolution that is infeasible, cannot be an optimal solution.

Definition 4.4.5 (Anti-monotonic)**.** A constraint c is *anti-monotonic* if for any state s, feature f and solution sets Θ_1 and Θ_2, with $s_1 = t(s, f)$, $s_2 = t(s, \bar{f})$, with $v_1 \in \Theta_1$ and $v_2 \in \Theta_2$, if $\neg c(v_1, s_1)$ or if $\neg c(v_2, s_2)$, then $v_1 \oplus v_2 \notin \text{opt}(\Theta, s)$.

Minimum support (minimum leaf node size) is an example of an anti-monotonic constraint (Nijssen and Fromont, 2010).

With these conditions and definitions in place, we have the necessary and sufficient conditions for separability and can present the main theoretical result of this chapter:

Theorem 4.4.6. *An optimization task* $o = \langle g, t, \succ, \oplus, c, s_0 \rangle$ *is* separable *if and only if its cost function* g *and transition function* t *are* Markovian*, its combining operator* $\oplus$ preserves order *over its comparison operator* $\succ$ *and the constraint* c *is* anti-monotonic*.*

4

4.4.3. Dynamic Programming Formulation

We now present the general DP framework *STreeD* (Separable Trees with Dynamic programming, pronounced as *street*):

$$T(s,d) = \begin{cases} \text{opt}\left(\bigcup_{\hat{k}\in\mathcal{K}}\{\ g(s,\hat{k})\ \}, s\right) & d = 0 \\ \text{opt}\left(\bigcup_{f\in\mathcal{F}} \text{merge}\left(T(t(s,f),d-1), T(t(s,\bar{f}),d-1), s, f\right), s\right) & d > 0 \end{cases} \tag{4.8}$$

Pseudo-code for STreeD and additional algorithmic techniques for speeding up computation, such as a special depth-two solver, caching, and upper and lower bounds, as well as techniques for sparse trees and hypertuning, can be found in Appendix 4.B. The appendix also provides the conditions for the use of these techniques. Furthermore, in Appendix 4.A we prove the following Theorem:

Theorem 4.4.7. *STreeD, as defined in Eq.* (4.8)*, finds the Pareto front for any optimization task that is* separable *according to Def. 4.4.2.*

4.4.4. Examples of Separable Optimization Tasks

To illustrate the flexibility of STreeD, we present four diverse example applications for which separable optimization tasks can be formulated. The first two are also covered by the *additivity* condition from (Nijssen and Fromont, 2010). The last two do not fit this condition but are covered by our framework. We cover related work for each example application in Appendix 4.C .

Cost-sensitive classification Cost-sensitive classification considers costs related to obtaining (measuring) the value of features in addition to misclassification costs (Lomax and Vadera, 2013). Typical applications are in the medical domain when considering expensive diagnostic tasks and asymmetrical misclassification costs.

Let $\mathcal{D}$ be the instances in a leaf node and let $M_{k,\hat{k}}$ be the misclassification costs when an instance with true label k is assigned label $\hat{k}$, and let $m(F, f)$ be the costs of obtaining the value of feature f (this value also depends on the set of previously measured features F, because we consider that some tests can be performed in a group and share costs). This results in the following cost function:

$$g(\langle \mathcal{D}, F \rangle, \hat{k}) = \sum_{(x,k) \in \mathcal{D}} M_{k,\hat{k}} \qquad g(\langle \mathcal{D}, F \rangle, f) = |\mathcal{D}| \cdot m(F, f) \tag{4.9}$$

This optimization task is separable because the function g is Markovian, the combination operator $\oplus$ is addition, and the comparison operator is $<$.

Prescriptive policy generation Decision trees can also be used to prescribe policies based on historical data, to maximize the expected reward (revenue) of the policy (Kallus, 2017; Bertsimas et al., 2019). An example is medical treatment assignment based on historical treatment response. This is done by reasoning over counterfactuals (the expected outcome if an action other than the historical action was taken).

We consider a dataset $\mathcal{D}$ containing instances (x, k, y) with $x \in \mathcal{X}$ the feature vector, $k \in \mathcal{K}$ the label or historically assigned treatment, and $y \in \mathbb{R}$ the observed outcome value. We assume no information on the historical treatment assignment policy. Let $Y(x, k)$ be the value outcome when assigning treatment k to an individual described by x. The goal now is to find a policy $\pi : \mathcal{X} \to \mathcal{K}$ that maximizes the expected outcome $Q(\pi) = \mathbb{E}[Y(x, \pi(x))]$.

Because not every treatment was applied to each person, we do not have full information on the values of $Y(x, k)$, and therefore we need a teacher model that provides information on the counterfactuals, based on the data that we do have. Similar to Jo et al. (2021) and Amram et al. (2022), we will here consider three teacher models: the direct method (DM), inverse propensity weighting (IPW), and the doubly robust approach (DR). The cost function for each of these is Markovian and the combining operator is addition. Therefore, prescriptive policy generation can be formulated as a separable optimization task.

Direct method The direct method (also called Regress and Compare) trains a teacher model $\hat{v}_k$ for the expected outcome y for each possible treatment k by splitting the training data based on the historical treatment. The objective value Q can therefore be formulated as follows:

$$Q_{\mathrm{DM}}(\mathcal{D}, \pi) = \frac{1}{|\mathcal{D}|} \sum_{(x,k,y) \in \mathcal{D}} \hat{v}_{\pi(x)}(x) \tag{4.10}$$

Inverse propensity weighting The propensity scores $\mu(x, k) = \mathbb{P}(k \mid x)$ give the probability of a treatment k for a given feature vector x. IPW implicitly models the counterfactuals by reweighing the actual data based on the inverse of their propensity scores. This requires the training of a propensity score

model $\hat{\mu}(x,k)$. Any ML model can be used to learn $\hat{\mu}$. Given a model for $\hat{\mu}$, the objective value Q of the policy becomes:

$$Q_{\text{IPW}}(\mathcal{D},\pi) = \frac{1}{|\mathcal{D}|} \sum_{(x,k,y)\in\mathcal{D}} \frac{\mathbb{1}(\pi(x)=k)}{\hat{\mu}(x,k)} \tag{4.11}$$

Doubly robust Finally, the doubly robust method (Dudík et al., 2011) combines IPW and DM as follows:

$$Q_{\text{DR}}(\mathcal{D},\pi) = \frac{1}{|\mathcal{D}|} \sum_{(x,k,y)\in\mathcal{D}} \left(\hat{v}_{\pi(x)}(x) + (y - \hat{v}_k(x)) \frac{\mathbb{1}(\pi(x)=k)}{\hat{\mu}(x,k)} \right) \tag{4.12}$$

4

Once the objective value function Q is known as stated above, this can be directly applied in our framework since each of the objectives $Q_{\text{IPW}}, Q_{\text{DM}}$ and Q_{DR} are separable, once the teacher model values resulting from $\hat{\mu}$ and $\hat{v}$ have been precomputed. For each of these functions, the combining operator is addition, the comparator is $>$ (maximization) and the cost function is determined by the value of the policy π_k that assigns treatment k to all individuals:

$$g(\mathcal{D},k) = Q(\mathcal{D},\pi_k) \tag{4.13}$$

It is also possible to add constraints to the policy, for example, when maximizing revenue under a capacity constraint. We show in Appendix 4.A.4 that capacity constraints are also separable.

Nonlinear classification metrics Nonlinear objectives such as F1-score and Matthews correlation coefficient are typically used for unbalanced datasets. Demirović and Stuckey (2021) provide a globally optimal DP formulation for optimizing such nonlinear objectives for binary classification. STreeD can also optimize for these metrics. For F1-score, for example, one can formulate a combined optimization task that measures the false positive rate for each class $k \in \{0,1\}$ as a separate optimization task:

$$g_k(\mathcal{D},\hat{k}) = \begin{cases} |\{(x,k') \in \mathcal{D} \mid k' \neq \hat{k}\}| & \hat{k} = k \\ 0 & \text{otherwise} \end{cases} \tag{4.14}$$

Both of these tasks are separable, and in Appendix 4.A.5 we show that multiple separable tasks can be combined into a new separable optimization task that results in the Pareto front for misclassifications for each class. This Pareto front can then be used to find the decision tree with, e.g., the best F1-score.

Group fairness Optimizing for accuracy can result in unfair results for different groups and therefore several approaches have been recommended to prevent discrimination (Mehrabi et al., 2021), one of which is *demographic parity* (Dwork et al., 2012), which states that the expected outcome for two classes should be the same. Another fairness metric is *equality of opportunity*, which requires that the

true positive rate for two classes should be the same. Previous MIP formulations for group fairness have been formulated (Aghaei et al., 2019; Jo et al., 2023) and recently also a DP formulation, even though at first hand a group fairness constraint does not seem separable (Van der Linden et al., 2022).

Demographic parity Let a denote a discrimination-sensitive binary feature and y a binary outcome, with $y = 1$ the preferred outcome and $\hat{y}$ the predicted outcome; then demographic parity can be described as satisfying

$$P(\hat{y} = 1 \mid a = 1) = P(\hat{y} = 1 \mid a = 0)\,. \tag{4.15}$$

Typically, this is simplified to limiting the difference between the probabilities to some small percentage δ. Let $N(a)$ be the number of people in group a, and $N(\bar{a})$ the people not in group a, and similarly, let $\hat{N}(y, a)$ be the number of group a receiving the positive outcome, and $\hat{N}(\bar{y}, a)$ the number of group a receiving the negative outcome, etc. Then Eq. (4.15) becomes:

$$\left| \frac{\hat{N}(y,a)}{N(a)} - \frac{\hat{N}(y,\bar{a})}{N(\bar{a})} \right| \leq \delta \tag{4.16}$$

By observing that $\hat{N}(\bar{y}, a)/N(a) = 1 - \hat{N}(y, a)/N(a)$, and $\hat{N}(\bar{y}, \bar{a})/N(\bar{a}) = 1 - \hat{N}(y, \bar{a})/N(\bar{a})$, Eq. (4.16) can be rewritten to the two constraints shown in the main text:

$$\frac{\hat{N}(y,a)}{N(a)} + \frac{\hat{N}(\bar{y},\bar{a})}{N(\bar{a})} \leq 1 + \delta \qquad \frac{\hat{N}(y,\bar{a})}{N(\bar{a})} + \frac{\hat{N}(\bar{y},a)}{N(a)} \leq 1 + \delta \tag{4.17}$$

The demographic parity requirement can now be formulated as two separable threshold constraints:

$$g_a(\mathcal{D}, \hat{k}) = \begin{cases} \frac{|\mathcal{D}_a|}{N(a)} & \hat{k} = 1 \\ \frac{|\mathcal{D}_{\bar{a}}|}{N(\bar{a})} & \hat{k} = 0 \end{cases} \qquad g_{\bar{a}}(\mathcal{D}, \hat{k}) = \begin{cases} \frac{|\mathcal{D}_a|}{N(a)} & \hat{k} = 0 \\ \frac{|\mathcal{D}_{\bar{a}}|}{N(\bar{a})} & \hat{k} = 1 \end{cases} \tag{4.18}$$

Equality of opportunity Similarly, we can optimize for other group concepts of fairness, such as *equality of opportunity*, which is satisfied when:

$$P(\hat{y} = 1 \mid y = 1, a = 1) = P(\hat{y} = 1 \mid y = 1, a = 0) \tag{4.19}$$

This can be computed by two separable threshold constraints:

$$g_a(\mathcal{D}, \hat{k}) = \begin{cases} \frac{|\mathcal{D}_{y,a}|}{N(y,a)} & \hat{k} = 1 \\ \frac{|\mathcal{D}_{y,\bar{a}}|}{N(y,\bar{a})} & \hat{k} = 0 \end{cases} \qquad g_{\bar{a}}(\mathcal{D}, \hat{k}) = \begin{cases} \frac{|\mathcal{D}_{y,a}|}{N(y,a)} & \hat{k} = 0 \\ \frac{|\mathcal{D}_{y,\bar{a}}|}{N(y,\bar{a})} & \hat{k} = 1 \end{cases} \tag{4.20}$$

In this equation $\mathcal{D}_{y,a}$ and $\mathcal{D}_{y,\bar{a}}$ are the number of instances with a positive outcome in the dataset for group a and $\bar{a}$, and $N(y, a)$ and $N(y, \bar{a})$ represent the number of group a and $\bar{a}$ that have the positive outcome in the data.

These cost functions with the two threshold constraints that limit both to $1 + \delta$, and an accuracy objective can be combined into one separable optimization task as shown in Appendix 4.A.4 and 4.A.5, and therefore it can be solved to optimality with STreeD. Note that we require group fairness on the whole tree and not on every subtree. In fact, requiring group fairness on every subtree is an example of a constraint that does not satisfy anti-monotonicity.

4.4.5. Comparison to Previous Theory and Methods

Dynamic programming theory In contrast to what is common in general DP theory, our theory does not assume solution values to be real valued (Karp and Held, 1967) or totally ordered (Elmaghraby, 1970). For example, Karp and Held (1967) formulate the *monotonicity* requirement of a DP as follows: for a given state s, action a, cost c and data p, let $h(c, s, a, p)$ be the cost of reaching state $t(s, a)$ through an input sequence that reaches state s with cost c. Monotonicity holds iff $c_1 \leq c_2$ implies $h(c_1, s, a, p) \leq h(c_2, s, a, p)$. In words: if a partial input sequence A_1 with cost c_1 dominates another input sequence A_2 with c_2, then any other input sequence that starts with A_1 dominates any other input sequence that starts with A_2, thus satisfying the principle of optimality. This notion is similar to our *order preservation*, but we do not assume that the costs are real valued or that the costs are totally ordered.

Similarly, Li and Haimes (1991) show how multiple optimization tasks can be combined into one, as we also show in Appendix 4.A.5. They construct a separable DP formulation for optimization tasks that can be deconstructed into a series of optimization tasks, each of which is separable and monotonic. Their method also returns the set of nondominated solutions. However, their theory assumes that the solution value for each sub-objective is real, completely ordered, and additive. This limits their theory to element-wise additive optimization tasks. Our theory does not share these limitations.

Dynamic programming approaches In comparison to the DP methods DL8 (Nijssen and Fromont, 2010) and GOSDT (Lin et al., 2020), STreeD covers a wider variety of objectives and constraints. Both STreeD and DL8 require constraints to be *anti-monotonic*. GOSDT does not support constraints. DL8 and GOSDT require *additivity*, whereas STreeD requires the less restrictive condition *order preservation* (see Def. 4.4.4). Moreover, STreeD can cover a range of new objectives by also allowing for objectives and constraints with a partially defined comparison operator, and by allowing to combine several such objectives and constraints. Examples of problems that could not be solved to global optimality by DL8 and GOSDT, but can be solved with our framework, are group fairness constraints, nonlinear metrics, and revenue maximization under a capacity constraint. Moreover, compared to DL8 and GOSDT, STreeD implements several algorithmic techniques for increasing scalability, such as a specialized solver for trees up to depth two (see Appendix 4.B).

Other approaches MIP can be used to model many optimization tasks, including non-separable optimization tasks. For example, optimizing decision tree policies for Markov Decision Processes can be optimized with MIP (Vos and Verwer, 2023), but

it is not clear how to do this with DP. However, MIP cannot deal with separable, but non-linear objectives, such as F1-score. Moreover, as our experiments also show, DP outperforms the scalability of MIP for the considered optimization tasks by several orders of magnitude.

CP so far has only been applied to maximizing accuracy (Verhaeghe et al., 2020) and (Max)SAT only to maximizing accuracy (Hu et al., 2020; Shati et al., 2023b) or finding the smallest perfect tree (Narodytska et al., 2018; Janota and Morgado, 2020).

Dunn (2018) proposes a framework that can optimize many objectives for decision trees (see also Bertsimas and Dunn (2019)). Because of MIP's scalability issues, they propose a local search method based on coordinate descent. However, this method does not guarantee a globally optimal solution. Moreover, their method is not able to find a Pareto front that optimizes multiple objectives at the same time.

4.5. Experiments

To show the flexibility of STreeD, we test it on the four example domains introduced in Section 4.4.4. Additionally, we compare STreeD with two state-of-the-art dynamic programming approaches for maximizing classification accuracy. STreeD is implemented in C++ and is available as a Python package.[1] All experiments are run on a 2.6 GHz Intel i7 CPU with 8GB RAM using only one thread. MIP models are solved using Gurobi 9.0 with default parameters (Gurobi Optimization LLC, 2022).

In our analysis, we focus on scalability performance in comparison to the state-of-the-art optimal methods (if available). We first discuss each of the five domains in detail and then conclude with a summary.

4.5.1. Cost-Sensitive Classification

Experiment setup

For cost-sensitive classification, we compare STreeD with TMDP (Maliah and Shani, 2021) both in terms of average expected costs and scalability. A direct fair comparison with TMDP is difficult, because TMDP provides no guarantee of optimality, allows for multi-way splits, and is run without a depth limit. We test STreeD and TMDP on three setups for 15 datasets from the literature and report normalized costs. We tune STreeD with a depth limit of $d = 4$. For each run, a timeout of 300 seconds is used. To our knowledge, no MIP methods for cost-sensitive classification including group discounts (i.e., when certain features are tested together, the feature cost decreases) exist, and therefore are not considered here.

We consider a constant but asymmetric misclassification costs, test costs (feature costs) that are constant or may depend on previous selected tests. For costs, we follow the naming conventions of Turney (2000). Misclassification costs or test costs that depend on individual cases or on feature values are easy extensions and also fit in the STreeD framework, but are not considered here.

[1] `https://github.com/AlgTUDelft/pystreed`

Dataset	$\lvert\mathcal{D}\rvert$	$\lvert\mathcal{F}_{disc}\rvert$	$\lvert\mathcal{F}_{bin}\rvert$	$\lvert\mathcal{K}\rvert$
Annealing	898	24	82	5
Breast	277	9	38	2
Car	1728	6	21	4
Diabetes	768	6	11	2
Flare	323	10	25	3
Glass	214	7	17	6
Heart	297	11	20	2
Hepatitis	154	16	20	2
Iris	150	4	12	3
Krk	28056	6	40	18
Mushroom	8124	21	111	2
Nursery	8703	8	26	5
Soybean	562	35	80	15
Tictactoe	958	9	27	2
Wine	178	13	32	3

Table 4.1: Datasets used in cost-sensitive classification. $|\mathcal{F}_{disc}|$ is the number of categorical features in the dataset as preprocessed by Lomax (2013). $|\mathcal{F}_{bin}|$ is the number of resulting binary features.

Data Table 4.1 lists the datasets used in this experiment (Janosi et al., 1988; Dua and Graff, 2017; Zwitter and Soklic, n.d.). These datasets were preprocessed and discretized according to the instructions by Lomax (2013). For methods that require binary features, the discretized features were binarized using one-hot encoding. Every dataset is split randomly 100 times in a train and test set, with 80% and 20% of the instances respectively.

We use the fixed feature costs as defined by Lomax (2013). The cost of misclassification strongly impacts the behavior of cost-sensitive algorithms. When misclassification costs are low, feature costs are comparatively high and branching is discouraged. Conversely, when misclassification costs are high, feature costs are comparatively low and therefore less important. In this case, overfitting also becomes more likely. Therefore, similar to the experiment setup by Lomax (2013) and Maliah and Shani (2021) we vary the misclassification costs and generate misclassification matrices with low, middle, and high misclassification costs. These matrices are defined based on the feature costs and the class frequency as follows: First, calculate the maximum possible feature costs C by summing the costs for all features. The default cost C_{def} for low, middle, and high-cost experiments are defined as $1/6\ C$, $1/3\ C$, and C respectively. By considering the relative frequency f_k of class k, the misclassification costs C_k for class k are defined as follows:

$$C_k = \frac{C_{def}}{f_k|\mathcal{K}|} \tag{4.21}$$

Normalized costs We report normalized costs, as defined by Turney (1995). The normalized costs are calculated by dividing the obtained costs by the standard costs. As before, let C be the sum of all feature costs, let f_k be the relative frequency of class k, and let $C_{k,\hat{k}}$ be the misclassification cost when an instance with true label k

is assigned label $\hat{k}$. Then the standard costs is defined as follows:

$$C + \min_k (1 - f_k) \cdot \max_{k,\hat{k}} C_{k,\hat{k}} \tag{4.22}$$

These standard costs are lower than the maximum possible costs, but function as a good upper bound on the average costs (Turney, 1995).

In one aspect we deviate from the original definition. In the original definition, f_k was calculated based on the whole dataset, but here it is based on the dataset that is examined (the train or test set).

Methods

Apart from STreeD, the following methods are evaluated in our experiments:

C4.5 (Quinlan, 1993b) is an extension of the ID3 algorithm (Quinlan, 1986) and uses top-down induction (TDI) based on an information gain measure. After construction, a pruning mechanism is used to remove branching nodes that are likely to result in overfitting. C4.5 is cost-insensitive and therefore functions as a baseline heuristic.

CSID3 (Tan, 1993) is a variant of ID3 with an updated splitting criterion. Instead of only information gain, it makes a trade-off between the information gain $I(f)$ and the cost $C(f)$ of feature f by the splitting criterion:

$$\frac{I(f)^2}{C(f)}$$

CSID3 does not consider varying misclassification costs.

EG2 (Núñez, 1991) is also a variant of ID3 with an updated splitting criterion:

$$\frac{2^{I(f)} - 1}{(C(f) + 1)^\omega}$$

The parameter ω, with values $0 \leq \omega \leq 1$, determines the weight of the cost factor. Following Turney (1995), we set $\omega = 1$. EG2 also does not consider varying misclassification costs.

IDX (Norton, 1989) is yet another variant of ID3 with the splitting criterion:

$$\frac{I(f)}{C(f)}$$

The look-ahead procedure of IDX is not implemented. IDX also does not consider varying misclassification costs.

MetaCost (Domingos, 1999) can be used to turn any error-based classifier into a cost-sensitive classifier. It iteratively fits a model, computes class probabilities per sample, and updates the training input labels such that the expected costs are minimized based on the class probabilities. Note that MetaCost only considers misclassification costs and ignores feature costs. We run MetaCost with 10 iterations.

TMDP (Maliah and Shani, 2021) formulates the problem as a Markov Decision Process (MDP). For every possible subset of features, it uses C4.5 to generate a normal cost-insensitive tree, and every leaf from the resulting tree is added as a state to the MDP. Each state is described by the path to the leaf. The state with the empty path is the starting state. In every state, the possible actions are to measure a feature or to classify a sample. If a feature is measured, the next state is the one described by the path that results from appending the measured feature to the current path. When classifying, the MDP moves to the terminal state. This MDP therefore is acyclic, and the values of the states can be computed in reverse topological order by using the training data. Based on these values, the tree can now be constructed by choosing the action with maximum value, beginning in the start node, and splitting into multiple nodes when selecting a feature to measure. This is repeated until all nodes are leaf nodes.

Because TMDP creates trees for every possible subset of the feature set, the total number of possible states explodes for an increasing number of features. Therefore, the authors propose that for a larger number of features, TMDP only tests all possible subsets of the 10 most costly features together with all the cheap features considered by default. In the analysis of TMDP here, the same approach is used.

Note that TMDP is the only method that is evaluated here that considers multi-way splits instead of binary splits.

For these methods, we use an implementation that is available online.[2] C4.5 is implemented in Weka, with CSID3, EG2, and IDX implemented as C4.5 but with the new splitting criterion. Leaf nodes do not select the majority class, but the class with the lowest total misclassification costs.

Results

Normalized costs Table 4.2 shows the normalized cost results for all methods on a variety of datasets. On 14 out of 15 datasets STreeD has the best performance or is not statistically significantly worse than the method with the best performance. The results show that TMDP and STreeD have similar performance and significantly outperform the other heuristics on all datasets except Flare. Therefore, the rest of the analysis will focus on TMDP and STreeD.

Table 4.3 shows the cost results for TMDP and STreeD for low, middle, and high misclassification costs. The difference in the results can be explained by several factors: multi-way splits versus binary splits, and the quality of the binarization. TMDP considers multi-way splits and therefore has a larger solution space, but does not necessarily find the best tree in this solution space according to the training data. STreeD, on the other hand, only considers binary decision trees and its solution quality depends on the binarization. However, it always finds the best binary decision tree for the training data.

[2] `https://github.com/shanigu/CostSensitive`

Dataset	C4.5	CSID3	EG2	IDX	MetaCost	TMDP	STreeD
Annealing	5.6	5.6	2.2	2.2	5.7	1.6	**1.2**
Breast	43.9	43.4	41.0	40.7	54.8	**21.0**	**20.4**
Car	43.1	42.2	39.8	39.9	42.2	16.1	**14.8**
Diabetes	50.2	44.5	43.4	43.4	55.9	**14.2**	**14.1**
Flare	17.9	**15.7**	**15.4**	**15.3**	21.8	18.0	16.7
Glass	37.7	35.6	30.1	29.0	39.4	**14.3**	**14.4**
Heart	45.4	23.0	20.0	19.9	44.7	**9.5**	**9.3**
Hepatitis	32.2	24.3	20.0	19.7	32.2	**16.2**	**15.6**
Iris	38.8	34.0	29.5	29.7	40.9	**12.8**	**12.8**
Krk	6.5	6.2	6.3	6.3	6.5	-	**1.7**
Mushroom	16.4	8.3	5.6	4.5	16.4	-	**1.9**
Nursery	0.4	0.4	0.4	0.4	0.4	0.1	**0.1**
Soybean	12.8	16.3	19.8	21.1	12.6	15.4	**11.4**
Tictactoe	45.0	45.0	45.0	45.0	45.0	**16.1**	**16.1**
Wine	25.0	18.8	18.1	17.5	24.8	12.2	**10.4**

Table 4.2: Out-of-sample normalized costs (%). Significantly ($p < 5\%$) best results are marked in bold. Timeouts are marked as '-'.

Runtime Figure 4.2 shows the difference in runtime for STreeD (hypertuned with $d = 4$) and TMDP. As can be seen, STreeD is often an order of magnitude faster than TMDP, and is significantly faster than TMDP for all datasets ($p < 5\%$), except for Iris, the smallest dataset.

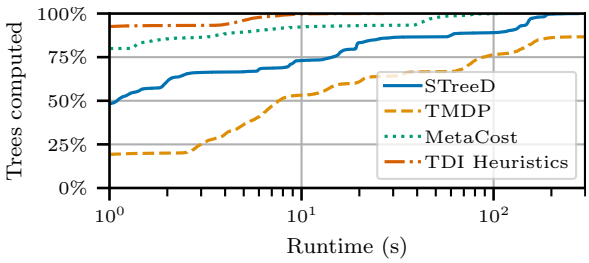


Figure 4.2: Runtime (s) comparison. The top-down induction heuristics (C4.5, CSID3, EG2, and IDX) have been grouped together as 'TDI Heuristics' because they almost do not differ in runtime. A timeout of 300 seconds is used. Note the logarithmic x-axis.

Interpretability Table 4.4 shows how the methods compare in terms of interpretability metrics. When only looking at the numbers, TMDP has the best score, and STreeD follows shortly after. The heuristics, as expected, score much worse. However, TMDP allows for multi-way split trees and the others do not. According to Piltaver et al. (2016), a higher branching factor increases the classifying time of an average user, although this effect can only be seen with branching factors higher than 3. Therefore, we can conclude that both TMDP and STreeD clearly outperform the heuristics in terms of interpretability.

	Low costs		Middle costs		High costs	
Dataset	STreeD	TMDP	STreeD	TMDP	STreeD	TMDP
Annealing	**1.3**	1.6	**1.2**	1.6	**1.2**	1.5
Breast	8.3	8.3	15.7	15.6	**37.3**	39.1
Car	9.8	9.8	**14.0**	15.8	**20.5**	22.9
Diabetes	7.3	7.2	**11.6**	11.8	23.4	23.5
Flare	9.9	9.8	**15.2**	16.3	**24.9**	28.0
Glass	10.9	10.8	13.8	14.1	18.5	18.1
Heart	4.0	4.0	7.2	7.5	16.7	17.1
Hepatitis	**8.4**	9.2	12.9	13.5	25.6	26.0
Iris	7.5	7.5	10.7	10.7	20.2	20.3
Krk	**1.8**	-	**1.8**	-	**1.7**	-
Mushroom	**1.7**	-	**1.8**	-	**2.1**	-
Nursery	0.1	0.1	0.1	0.1	0.1	0.1
Soybean	**10.9**	18.7	**11.6**	16.3	11.8	**11.3**
Tictactoe	6.8	6.8	12.7	12.7	28.8	28.9
Wine	**8.0**	11.3	**11.9**	12.5	**11.2**	12.8

Table 4.3: Test normalized cost (%) for varying misclassification costs. Significantly ($p < 5\%$) best results are marked in bold. Timeouts are marked as '-'.

	C4.5	CSID3	EG2	IDX	MetaCost	TMDP	STreeD
Tree depth	7.9	7.1	7.1	7.1	7.2	**1.9**	2.2
Leaf nodes	26.1	27.2	28.6	28.7	24.8	7.4	**5.4**
Question length	4.4	4.7	5.0	5.0	4.7	**1.7**	2.8

Table 4.4: Interpretability metrics. Significantly ($p < 5\%$) best results are marked in bold. Datasets for which TMDP exceeded the timeout limit are ignored.

Discussion

The results above clearly show the advantage of TMDP and STreeD over the heuristics. For most datasets, the cost performance of TMDP and STreeD was much better than the heuristics. This is best explained by the fact that the splitting criteria of the heuristics do not directly capture the objective, and in these cases even ignore the misclassification costs.

Comparing TMDP and STreeD is more difficult. In terms of costs, both at times perform best, although on average STreeD's out-of-sample costs are (significantly) lower: 13.3% normalized costs versus TMDP's 14.3%. Both methods show some overfitting and this could be further addressed. Direct comparison is difficult though, because they differ in many ways: TMDP has categorical data with multi-way splits and seeks a tree without a depth limit. STreeD has binary data with binary splits and seeks the best tree up to some maximum size.

The runtime of STreeD is at least an order of magnitude lower than TMDP. One of the consequences here is that TMDP was not able to find good trees for several datasets within the given time limit. Moreover, these results are produced with

TMDP's limit of considering only subsets for the 10 most costly features. Without this limit, TMDP's runtime would have been worse. STreeD therefore is more scalable than TMDP.

Both STreeD and TMDP have good interpretability and clearly outperform the heuristics. The heuristics' trees are often too large to be considered interpretable.

From these results it can be concluded that STreeD can successfully be applied to cost-sensitive classification, outperforming the state of the art in cost performance and runtime, while providing small interpretable trees.

4.5.2. Prescriptive Policy Generation

Experiment setup

We compare our method with the MIP model Jo-PPG-MIP (Jo et al., 2021) and the recursive tree search algorithm CAIPWL-opt (Zhou et al., 2022). Jo et al. (2021) show both analytically and through experiments the benefit of their method over Bertsimas et al. (2019) and Kallus (2017). Their method constructs optimal binary trees based on three different teacher models: regress and compare, inverse propensity weighting, and a doubly robust method. CAIPWL-opt constructs optimal binary trees based on the doubly robust method. Our method implements the same teacher models; thus, all methods find the same solution. Therefore, we compare here only the runtime performance.

Jo-PPG-MIP Jo et al., 2021 is a MIP model based on a max-flow formulation for optimal decision trees from Aghaei et al. (2024) but with the aforementioned Q-function as the objective. We call this method Jo-PPG-MIP, after their first author. Our implementation of Jo-PPG-MIP is based on the open-source code for Aghaei et al. (2024), for which we updated the objective.[3]

CAIPWL-opt Zhou et al., 2022 also observe the scalability issues with MIP-based methods and therefore propose a recursive tree-search algorithm that maximizes the doubly robust metric. Their method does not use caching, and therefore this method does not qualify as a dynamic programming approach. They suggest their optimal method is best for trees of at most depth three. For deeper trees, they suggest using approximate methods. In our experiments we use their publicly available R-package.[4]

As further detailed below, we test on the Warfarin dataset (International Warfarin Pharmacogenetics Consortium, 2009) and generate 225 training scenarios with 3367 instances and 29 binary features and run each algorithm with a given maximum depth of $d \in [1, 5]$. We also generate 275 synthetic datasets with 10, 100, and 200 binary features, each with 500 training samples, and run each algorithm on every scenario with a given maximum depth of $d = 1, 2$ and 3.

[3] `https://github.com/pashew94/StrongTree`
[4] `https://github.com/grf-labs/policytree`

	$\phi(x)$	$\kappa(x)$
$f = 2$	$\frac{1}{2}x_1 + x_2$	$\frac{1}{2}x_1$
$f = 10$	$\frac{1}{2}\sum_{j=1}^{2} x_j + \sum_{j=3}^{6} x_j$	$\sum_{j=1}^{2} \max(0, x_j)$
$f = 20$	$\frac{1}{2}\sum_{j=1}^{4} x_j + \sum_{j=5}^{8} x_j$	$\sum_{j=1}^{4} \max(0, x_j)$

Table 4.5: Mean effect and treatment effect functions for the synthetic dataset, based on (Athey and Imbens, 2016).

Synthetic data

For the experiment with synthetic data, we follow the setup as done by Athey and Imbens (2016) and Jo et al. (2021). We consider two possible treatments $K = \{0, 1\}$, input vectors X, which consist of f independent variables with distribution $N(0, 1)$ and for each X_i corresponding potential outcomes $Y_i(k)$:

$$Y_i(k) = \phi(X_i) + \frac{1}{2}(2k - 1) \cdot \kappa(X_i) + \varepsilon_i. \tag{4.23}$$

In this equation, $\phi(x)$ is the mean effect, $\kappa(x)$ is the treatment effect, and $\varepsilon_i \sim N(0, 0.1)$ is independent noise. Based on Athey and Imbens (2016) we consider three variations of ϕ and κ, for $f = 2, 10, 20$, as shown in Table 4.5. As can be seen from this table, for $f = 2$, the mean effect ϕ depends on both variables, and the treatment effect only on one. For $f = 10$ and $f = 20$, the mean effect depends on a subset of the variables. The treatment effect also depends on a subset of the variables, but only when those values are positive. Moreover, for these two cases, some variables are not related to the potential outcomes at all and function as noise in the dataset. For the training dataset, per instance, we then select the optimal treatment (the treatment with the maximum value for Y_i) as the historical treatment with probability $p \in \{0.1, 0.25, 0.5, 0.75, 0.9\}$ and store the corresponding outcome in the dataset. We repeat this process five times and generate training sets and test sets with sizes 500 and 10000 respectively.

We binarize each of the original features into 10 bins with a similar frequency count. The teacher models are then trained on the original data. For IPW we train a decision tree model (dt, computed with CART), logistic regression (log), and the true propensity scores. For DM we train a linear regression (lr) and a lasso regression with $\alpha = 0.08$. For DR we use all six combinations of the IPW and DM methods.

Runtime The setup above results in 75 scenarios that are solved with 11 different teacher methods. We ran all three methods with a maximum tree depth of 1, 2, and 3, resulting in 2475 problems for each method to solve.

Figure 4.3 shows the runtime performance by listing how many instances of the synthetic dataset can be solved within the specified amount of runtime. Note

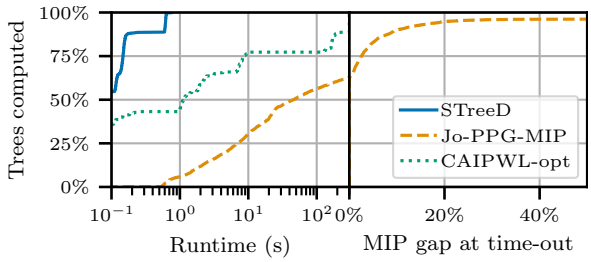


Figure 4.3: Proportion of instances that can be solved per method for the synthetic datasets within a given runtime (s). Note the logarithmic x-axis for runtime.

	Jo		CAIPWL-opt		STreeD	OOS Correct
d	Timeout	Time (s)	Timeout	Time (s)	Time (s)	Treatment
1	-	30	-	< 1	< 1	83.1%
2	62%	215	-	5	< 1	84.6%
3	100%	-	-	96	< 1	86.6%
4	100%	-	100%	-	< 1	88.0%
5	100%	-	100%	-	3	89.7%

Table 4.6: Results for the Warfarin dataset. The reported runtime is the average of runs that did not result in a timeout (300s).

that this is only the time required to build the tree. It does not include the time to train the teacher model. These results show that STreeD is several orders of magnitude faster than Jo-PPG-MIP and also scales much better than CAIPWL-opt while providing exactly the same solutions.

Warfarin dosing

Similar to previous studies, we test our method on the Warfarin dosing use-case (Kallus, 2017; Bertsimas et al., 2019; Jo et al., 2021). Warfarin is an anticoagulant, but the appropriate dose may vary up to a factor 10 among patients. Based on experimental tests, a dataset is compiled which prescribes for 5700 patients a suggested dose (International Warfarin Pharmacogenetics Consortium, 2009). For our experiment, unless otherwise specified, we follow the setup by Jo et al. (2021).

We calculate the square root of the suggested weekly dose by the formula presented in the supplementary material section S1e of International Warfarin Pharmacogenetics Consortium (2009) and add noise of $N(0, 0.02)$. This number is then squared and divided by seven to give the suggested daily dose. The features used in their formula are also the features used for the dataset, preprocessed as specified by them. The only three non-binary features (age, weight, and height) are binarized by splitting them into five buckets with approximately the same number of instances. Instances with missing values for weight, height, age, and the therapeutic dose of Warfarin are removed. This leaves 4490 instances. The daily suggested dose is discretized into three groups, with $k^{opt} = 0$ if the dose is less than 3 mg/day, $k^{opt} = 1$ if the dose

is between 3 and 7 mg/day, and $k^{opt} = 2$ if the dose is larger than 7 mg/day. The observed outcome is $Y_i(k_i) = 1$, if $k_i = k_i^{opt}$; and $Y_i(k_i) = 0$ otherwise.

For simulating historical data, three different setups are tested. In the first, every individual receives a uniformly random treatment k_i in the historical data with the corresponding outcome Y_i, to simulate a marginally randomized setting. In the second and third setups, historical treatments are obtained by using the formula for the weekly dose from International Warfarin Pharmacogenetics Consortium (2009) mentioned above with each coefficient in this formula perturbed by a random amount from $U(-0.06, 0.06)$ and $U(-0.11, 0.11)$ respectively, to change the probability of the historical data having the correct assignment. Each setup is used five times to generate historical treatments. Each of these 15 datasets is then randomly split five times into a training and test set with 75% and 25% of the instances respectively. For each of these, a decision tree is trained for IPW scores and a random forest regressor for DM estimates.

Results Table 4.6 shows the results for applying Jo-PPG-MIP and STreeD to the Warfarin case. The results are very similar for the different teacher methods and the probability of having the correct assignment in the historical data.[5] Therefore we just show the average results for trees of depth 1-5.

These results show that STreeD is several orders of magnitude faster than both Jo-PPG-MIP and CAIPWL-opt. Because of this, STreeD can find optimal trees for larger depth, which for this problem results in a better out-of-sample (OOS) correct treatment rate.

4.5.3. Nonlinear Classification Metrics

For assessing STreeD's performance on optimizing nonlinear metrics such as F1-score, we compare it with the existing nonlinear MurTree method (Demirović and Stuckey, 2021). We tasked both methods to find optimal trees of depth 3-4 according to an F1-score metric for 25 binary classification datasets from the literature. F1-score is the harmonic mean of precision and recall and can be calculated as follows based on the number of true positives (tp), false positives (fp), and false negatives (fn): $tp/(tp + 0.5(fp + fn))$.

Setup We compare STreeD with the MurTree algorithm by Demirović and Stuckey (2021) on a set of binarized datasets with binary class as described in Table 4.7. The datasets are originally from the UCI repository (Dua and Graff, 2017) and from (Turing Institute (Glasgow), 1987; Janosi et al., 1988; Zwitter and Soklic, 1988a,b; Mangasarian and Wolberg, 1990; Angwin et al., 2016a; FICO et al., 2018). We use the binarized datasets from the MurTree repository.[6] For each dataset, both algorithms need to find a tree of depth three and four that optimizes F1-score. We repeat the tests five times and report the average runtime.

[5] This is different from the results in (Jo et al., 2021). We have corresponded with the authors about this but were not able to reproduce their results. We will include the test setup in our repository to enable reproducibility of the results.

[6] `https://bitbucket.org/EmirD/MurTree-bi-objective`

					$d=3$		$d=4$	
Dataset	$\lvert\mathcal{D}\rvert$	$\lvert\mathcal{D}^+\rvert$	$\lvert\mathcal{D}^-\rvert$	$\lvert\mathcal{F}\rvert$	MurTree	STreeD	MurTree	STreeD
Anneal	812	625	187	93	< 1	$\mathbf{< 1}$	24	**2**
Audiology	216	57	159	148	< 1	$\mathbf{< 1}$	54	**2**
Australian-Credit	653	357	296	125	2	$\mathbf{< 1}$	139	**19**
Breast-Wisconsin	683	444	239	120	< 1	$\mathbf{< 1}$	61	**7**
COMPAS	6907	3196	3711	12	2	$\mathbf{< 1}$	7	$\mathbf{< 1}$
Diabetes	768	500	268	112	3	$\mathbf{< 1}$	117	**13**
Fico	10459	5000	5459	17	6	$\mathbf{< 1}$	26	**3**
German-Credit	1000	700	300	112	6	$\mathbf{< 1}$	164	**34**
Heart-Cleveland	296	160	136	95	< 1	$\mathbf{< 1}$	43	**6**
Hepatitis	137	111	26	68	< 1	$\mathbf{< 1}$	6	$\mathbf{< 1}$
Hypothyroid	3247	2970	277	88	< 1	$\mathbf{< 1}$	28	**3**
Ionosphere	351	225	126	445	26	**11**	-	-
Kr-vs-kp	3196	1669	1527	73	< 1	$\mathbf{< 1}$	22	**3**
Letter	20000	813	19187	224	46	**7**	-	**469**
Lymph	148	81	67	68	< 1	$\mathbf{< 1}$	8	**1**
Mushroom	8124	4208	3916	119	< 1	$\mathbf{< 1}$	$\mathbf{< 1}$	< 1
Pendigits	7494	780	6714	216	13	**4**	-	**240**
Primary-Tumor	336	82	254	31	< 1	$\mathbf{< 1}$	< 1	$\mathbf{< 1}$
Segment	2310	330	1980	235	< 1	< 1	$\mathbf{< 1}$	< 1
Soybean	630	92	538	50	< 1	$\mathbf{< 1}$	3	$\mathbf{< 1}$
Splice-1	3190	1655	1535	287	17	**9**	-	-
Tic-Tac-Toe	958	626	332	27	< 1	$\mathbf{< 1}$	1	$\mathbf{< 1}$
Vehicle	846	218	628	252	5	**2**	-	**130**
Vote	435	267	168	48	< 1	$\mathbf{< 1}$	3	**1**
Yeast	1484	463	1021	89	5	$\mathbf{< 1}$	68	**8**

Table 4.7: Runtime (s) for minimizing biobjective misclassification score for two classes, with timeouts beyond 600s denoted with '-'. Best results are shown in bold. Average of five runs.

Results Table 4.7 shows how STreeD compares to MurTree on optimizing nonlinear metrics. By using the geometric mean of the relative performance (Fleming and Wallace, 1986), we can see that for trees of depth three, STreeD is on average 7.1 times faster than MurTree; for trees of depth four, it is on average 6.8 times faster. This performance difference can be partly explained by the new upper and lower bound technique, as explained in Appendix 4.B.

This same method could also be used to optimize other nonlinear metrics that are monotonic with respect to the false positive and negative scores, such as Matthews correlation coefficient.

4.5.4. Group Fairness

We here consider two versions of *group fairness*: demographic parity and equality of opportunity (Mehrabi et al., 2021; Caton and Haas, 2023). In our experiments we compare STreeD with the two MIP methods Aghaei-Fair-MIP (Aghaei et al., 2019) and Jo-Fair-MIP (Jo et al., 2023); and the DP method DPF (Van der Linden et al.,

	Aghaei		Jo		DPF		STreeD	
Dataset	d=2	d=3	d=2	d=3	d=2	d=3	d=2	d=3
Adult	-	-	-	-	< 1	< 1	< 1	2
Bank	-	-	-	-	< 1	3	< 1	4
Com.&Crime	-	-	-	-	< 1	5	< 1	29
COMPAS r.	-	-	-	-	< 1	< 1	< 1	< 1
COMPAS v.r.	-	-	-	-	< 1	< 1	< 1	< 1
Dutch	-	-	-	-	< 1	6	< 1	8
German	-	-	-	-	< 1	2	< 1	62
KDD	-	-	-	-	1	39	3	26
OULAD	-	-	-	-	< 1	3	< 1	19
Ricci	3	44	2	13	< 1	< 1	< 1	< 1
Stud. Math	-	-	473	-	< 1	< 1	< 1	2
Stud. Port.	-	-	-	-	< 1	< 1	< 1	3

Table 4.8: Runtime (s) results for optimizing with a *demographic parity* constraint. Timeouts (>600s) are marked with '-'.

			STreeD	
Dataset	$\|\mathcal{D}\|$	$\|\mathcal{F}\|$	$d = 2$	$d = 3$
Adult	45222	17	< 1	1
Bank	45211	46	< 1	7
Com.&Crime	1994	97	< 1	11
COMPAS r.	6172	9	< 1	< 1
COMPAS v.r.	4020	9	< 1	< 1
Dutch	60420	58	< 1	7
German	1000	69	< 1	50
KDD	284556	117	3	37
OULAD	21562	45	< 1	19
Ricci	118	4	< 1	< 1
Stud. Math	395	55	< 1	< 1
Stud. Port.	649	55	< 1	1

Table 4.9: Runtime (s) results for optimizing with an *equality of opportunity* constraint. Timeouts (>600s) are marked with '-'.

2022). The comparison focuses on scalability.

Setup We follow the setup by Van der Linden et al. (2022), by testing each method on 12 datasets, preprocessed and binarized as described by Le Quy et al. (2022). Categorical variables are binarized using one-hot encoding (variables with twenty or more categories are removed). For each dataset, the task is to find an optimal decision tree of depth 2-3 with at most 1% discrimination. A timeout of 600 seconds is used. Tests are repeated five times and we report averages. For all methods, we show results for enforcing a demographic parity constraint. For STreeD, we also

show runtime results for optimizing with an equality-of-opportunity constraint.

References to the original datasets can be found here (Dutch Central Bureau for Statistics, 2001a; Cortez and Silva, 2008; Moro et al., 2014; Strack et al., 2014; Angwin et al., 2016a; Dua and Graff, 2017; Kuzilek et al., 2017).

Results Table 4.8 shows the runtime results. Both MIP methods almost always hit the time limit, except for the smallest datasets. In contrast, both DP methods find optimal decision trees for depth two in less than one second except for the KDD-Census income dataset. For depth three DPF outperforms STreeD. This is partly because of how the fairness constraint is formulated for STreeD: as two threshold constraints. For better performance, an objective more like the one presented in DPF could be formulated for STreeD. However, formulating and implementing a fairness constraint in STreeD is much easier than developing a specialized method such as DPF. For equality of opportunity, STreeD shows similar results as when optimizing demographic parity.

In summary, STreeD shows performance not too far off from an existing specialized DP method; allows for easy formulation of new types of constraints, and outperforms both MIP methods by a large margin.

4.5.5. Classification Accuracy

The most studied use case for optimal decision trees is maximizing accuracy. Since STreeD generalizes previous dynamic programming approaches for optimal decision trees, here we evaluate how STreeD performs compared to two state-of-the-art dynamic programming approaches to optimal decision trees: DL8.5 and MurTree.

DL8.5 (Aglin et al., 2020a,b) is a dynamic programming approach that improves on DL8 (Nijssen and Fromont, 2007, 2010) by adding branch-and-bound and new caching techniques. We use their publicly available Python package.[7] Note that we compared to their latest available version, which incorporates ideas from MurTree (Demirović et al., 2022).

MurTree (Demirović et al., 2022) is also a dynamic programming approach for optimal decision trees. Like DL8.5 it incorporates branch-and-bound and caching. Other improvements are a similarity lower bound and a special depth-two solver. We use the publicly available Python package.[8]

We test both methods and STreeD on a set of binarized datasets with binary class as described in Table 4.10. These are the same datasets as thosed in the nonlinear metric comparison above. For each dataset, all three algorithms need to find a tree of depth four or five with minimum misclassification score. We repeat the tests five times and report the average runtime in Table 4.10. Runtime results are also shown in Fig. 4.4.

By using the geometric mean of the relative performance (Fleming and Wallace, 1986), we can see that MurTree on average is 1.25 times faster than STreeD and

[7] `https://github.com/aia-uclouvain/pydl8.5`
[8] `https://github.com/MurTree/pymurtree`

			$d = 4$			$d = 5$		
Dataset	$\|\mathcal{D}\|$	$\|\mathcal{F}\|$	DL8.5	MurTree	STreeD	DL8.5	MurTree	STreeD
Anneal	812	93	2	**< 1**	< 1	17	**2**	5
Audiology	216	148	4	< 1	**< 1**	< 1	**< 1**	**< 1**
Australian-Credit	653	125	23	**1**	2	-	**21**	37
Breast-Wisconsin	683	120	4	**< 1**	1	15	1	7
Diabetes	768	112	24	**1**	2	-	**22**	34
Fico	10459	17	**< 1**	< 1	< 1	**< 1**	1	2
German-Credit	1000	112	36	**2**	3	-	61	94
Heart-Cleveland	296	95	6	**< 1**	< 1	80	**3**	7
Hepatitis	137	68	< 1	**< 1**	< 1	2	< 1	**< 1**
Hypothyroid	3247	88	5	1	**< 1**	94	13	**11**
Ionosphere	351	445	-	**69**	152	-	146	**112**
Kr-vs-kp	3196	73	2	< 1	**< 1**	18	6	**4**
Letter	20000	224	547	200	**145**	-	-	-
Lymph	148	68	1	**< 1**	< 1	< 1	< 1	**< 1**
Mushroom	8124	119	3	< 1	**< 1**	1	< 1	**< 1**
Pendigits	7494	216	160	60	74	-	240	**109**
Soybean	630	50	< 1	**< 1**	< 1	6	**< 1**	1
Splice-1	3190	287	-	**148**	156	-	-	-
Tic-Tac-Toe	958	27	< 1	**< 1**	< 1	2	**< 1**	< 1
Vehicle	846	252	79	8	15	-	**137**	343
Vote	435	48	< 1	**< 1**	< 1	5	**< 1**	1
Yeast	1484	89	10	**< 1**	1	280	**13**	18
Average rank			3.0	**1.29**	1.71	2.89	**1.39**	1.72

Table 4.10: Runtime (s) for maximizing accuracy, with timeouts beyond 600s denoted with '-'. Best results are shown in bold. Average of five runs. Datasets for which all methods finish within one second are left out.

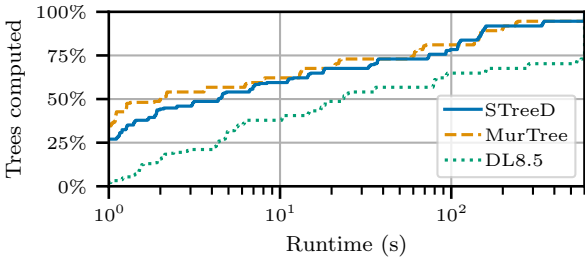
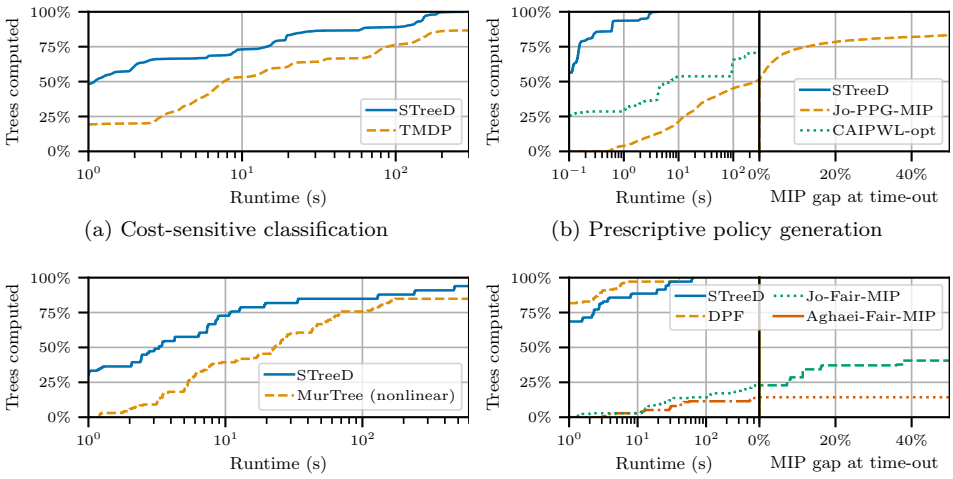


Figure 4.4: Runtime (s) comparison. A timeout of 300 seconds is used. Note the logarithmic x-axis.

STreeD on average is 6.4 times faster than DL8.5. According to a Wilcoxon signed rank test with a significance level 5%, both results are significant.

Since STreeD outperforms DL8.5 and DL8.5 outperforms DL8, we can conclude that STreeD also outperforms DL8 in terms of scalability. Furthermore, Demirović et al. (2022) compared MurTree with GOSDT (Lin et al., 2020) and observed that



(a) Cost-sensitive classification

(b) Prescriptive policy generation

(c) Nonlinear classification metrics: F1-score

(d) Group fairness: Demographic parity

Figure 4.5: The percentage of instances for which the algorithm returns the (optimal) tree within the given runtime (higher is better). When MIP methods hit the timeout, the plot shows which percentage of problems are solved within the reported MIP gap. Instances for which all methods return a tree within one second are left out. Note the logarithmic x-axis.

GOSDT resulted in a timeout for 65% of the 68 datasets when computing optimal trees with maximum depth four, whereas MurTree did not result in a single timeout. Since STreeD's performance is only a factor of 1.25 slower than MurTree, we conclude that STreeD also outperforms GOSDT.

4.5.6. Summary of the Results

Fig. 4.5 summarizes the results listed above for four of the application domains.

Cost-sensitive classification For 21 out of 45 scenarios STreeD generates trees with significantly lower average out-of-sample costs than TMDP (p-value $< 5\%$). TMDP has lower costs in 1 out of 45 scenarios. In all other scenarios, there is no significant difference. Both methods return trees that perform better and are smaller than those constructed by heuristics. Fig. 4.5a shows that STreeD is significantly faster than TMDP. Thus, it can be concluded that STreeD on average has slightly better out-of-sample performance than TMDP, returns trees of similar size, and is significantly faster than TMDP.

Prescriptive policy generation Fig. 4.5b shows that STreeD can find optimal solutions orders of magnitude faster than both Jo-PPG-MIP and CAIPWL-opt while obtaining the same performance in maximizing the expected outcome.

Nonlinear classification metrics Fig. 4.5c shows that STreeD, while being more general, is on average seven times faster than MurTree (computed with geometric mean). This performance gain is in part due to a more efficient computation of the

lower and upper bounds, as explained in Appendix 4.B.

This shows that STreeD can be applied to nonlinear objectives and even improves on the performance of a state-of-the-art specialized DP method for this task.

Group fairness Fig. 4.5d shows the runtime results for group fairness. As expected, DPF performs best since it is a DP method specifically designed for this task. STreeD is more general and yet remains close to DPF, while being several orders of magnitude faster than both MIP methods.

It can be concluded that STreeD allows for easy modeling of complex constraints such as demographic parity and equality of opportunity while outperforming MIP methods by a large margin.

Classification accuracy On average STreeD is more than six times faster than DL8.5 and is close to MurTree, with MurTree being 1.25 faster (computed with the geometric mean). Since DL8.5 outperforms DL8 (Aglin et al., 2020a) and MurTree outperforms GOSDT by a large margin (Demirović et al., 2022), we can conclude that STreeD not only allows for a broader scope of optimization tasks than DL8 and GOSDT, but is also more scalable.

4.6. Conclusion

We present STreeD: a general framework for finding optimal decision trees using dynamic programming (DP). STreeD can optimize a broad range of objectives and constraints, including nonlinear objectives, multiple objectives, and non-additive objectives, provided that these objectives are separable. We apply DP theory to provide necessary and sufficient requirements for such separability in decision tree optimization tasks and introduce the new notion order preservation that guarantees the principle of optimality (Bellman, 1957). We show that these results are more general than the state-of-the-art general DP solvers for decision trees (Nijssen and Fromont, 2010; Lin et al., 2020).

We illustrate the generalizability of STreeD in five application domains, two of which are not covered by the previous general DP solvers. For each, we present a separable optimization formulation and compare STreeD's scalability to the domain-specific state of the art. The results show that STreeD sustains or improves the scalability performance of DP methods, while being flexible, and outperforms mixed-integer programming methods by a large margin on these example domains.

STreeD allows for a much broader set of applications than presented here. Therefore, future work should further investigate STreeD's performance on, for example, regression tasks and other non-additive objectives. STreeD could also be used to further develop optimization of trees that lack explanation redundancy Izza et al., 2022. Finally, STreeD could further be improved by adding multi-way branching and allowing for continuous features.

Appendices for Chapter 4

4.A. Proofs

The following sections provide proofs for the propositions and theorems from the main text. We first prove Prop. 4.4.3 that claims that an optimization task is separable if it has a Markovian cost and transition function and if it satisfies the principle of optimality. We then prove Theorem 4.4.6 about the necessary and sufficient conditions for an optimization task to be separable. Next, we prove Theorem 4.4.7 about STreeD being able to solve separable optimization tasks to optimality. We extend the theory of the main text by introducing anti-monotonic threshold constraints. Finally, we show how multiple separable optimization tasks can be combined into a new separable optimization task.

4.A.1. Proof of Prop. 4.4.3 about Separability

Def. 4.4.2 defines separability by stating that for a given optimization task the optimal solution to any subtree in a tree search should be independent of any of the decision variables other than the decision variables of the subtree search and the parent node branching decisions. Prop. 4.4.3 states that this is the same as saying that the optimization task has a Markovian cost and transition function and satisfies the principle of optimality (Bellman, 1957), i.e., optimal solutions can only be constructed from optimal solutions to subproblems:

Definition 4.A.1 (Principle of Optimality)**.** Let $o = \langle g, t, \succ, \oplus, c, s_0 \rangle$ be an optimization task, let s be a state, let f be a branching decision, and let Θ_1 and Θ_2 be all the solutions to subproblems with state $t(s, f)$ and $t(s, \bar{f})$, and let $\Theta = \text{merge}(\Theta_1, \Theta_2, s, f)$. Then optimization task o satisfies the *principle of optimality* if and only if for every state s, branching decision f and solution sets Θ_1 and Θ_2, the following holds:

$$\text{opt}\left(\Theta, s\right) = \text{opt}\left(\text{merge}(\text{opt}\left(\Theta_1, t(s, f)\right), \text{opt}\left(\Theta_2, t(s, \bar{f})\right), s, f)\right) \tag{4.24}$$

With this definition in place, we can prove Prop. 4.4.3.

Proof. Prop. 4.4.3 requires two conditions: a Markovian cost and transition function, and the principle of optimality. To prove Prop. 4.4.3, we need to show that both conditions are necessary and sufficient. Therefore, we first show that if both conditions hold, the optimization task must be separable. We then show that if one of the two conditions of Prop. 4.4.3 fails, the optimization task is not separable.

Sufficient: To show that the two conditions of Prop. 4.4.3 are sufficient we use total induction over the tree depth d.

The base step ($d = 0$), considers the search for optimal leaf nodes. If a Markovian cost function $g(s, \hat{k})$ exists, the optimal label assignment can be determined by $\text{opt}\left(\cup_{\hat{k} \in \mathcal{K}} g(s, \hat{k}), s\right)$. This does not depend on any decision variable other than the state s and the leaf node label assignment $\hat{k}$ and therefore Def. 4.4.2 is satisfied for all tree searches up to depth zero.

The induction step (for $d > 0$) assumes Def. 4.4.2 is satisfied for all tree searches up to depth $d-1$. For these tree searches, optimal solutions can now be constructed by iterating over all possible branching decisions for the root node, using Eq. (4.24) to find optimal solutions when the root node branching decision is fixed, and using opt to select the optimal solution from the union of all these solution sets. Since neither the subtree search, nor Eq. (4.24), nor the transition function, nor the branching cost function, nor the union operator, nor the opt function depend on any decision variable other than the decision variables for this tree search and the parent node decisions, the optimal solution for this search can also be constructed independently of those variables.

Necessary conditions: If an optimization task is separable, a Markovian cost function and transition function necessarily exist. Since a node's state is the product of applying the transition function on the previous state, this state can be considered as a function of the initial state s_0, the parent nodes' branching decisions F, and the immediate action: the branching decision or the label assignment. Now assume that no Markovian cost and transition function exist. This means that the cost of an action cannot be expressed in terms of only the state, the immediate action, and the parent's branching decisions. This means that Def. 4.4.2 is not satisfied: a contradiction. Therefore a Markovian cost and transition function necessarily exists if an optimization task is separable.

For tree searches with a depth larger than zero, optimal solutions should be able to be constructed from optimal solutions to subtree searches, as defined in Eq. (4.24). Consider such a subtree search, which after deciding on the branching decision in its root node, creates two subproblems for the resulting children nodes. The optimal solution should be constructible from the optimal solution to these subproblems according to Eq. (4.24). If the optimal solution to this tree search cannot be constructed from the optimal solution to its subtree searches, then the optimal solutions to the subtree searches are not optimal: a contradiction. Therefore, Eq. (4.24) is a necessary condition. □

4.A.2. Proof of Theorem 4.4.6: Conditions for Separability

Theorem 4.4.6 states that an optimization task is separable if its cost and transition functions are Markovian, its combining operator preserves order over the comparison operator and its constraint is anti-monotonic.

To prove that these conditions are both sufficient and necessary, we need to show that the conditions mentioned in Theorem 4.4.6 are equivalent to those mentioned in Prop. 4.4.3. We will first show that these conditions are sufficient, and then that they are also necessary.

Proof. Let $o = \langle g, t, \succ, \oplus, c, s_0 \rangle$ be an optimization task, for which it holds that the cost function g and transition function t is Markovian, the combining operator $\oplus$ preserves order over $\succ$ and the constraint c is anti-monotonic. Furthermore, since the comparator $\succ$ is used to determine Pareto dominance, we may assume that it is *transitive*.

We will call an optimization task that satisfies Eq. (4.24) *splittable*. We begin by

proving that o is splittable. Consider any state s, any branching feature f, such that $s_1 = t(s, f)$ and $s_2 = t(s, \bar{f})$, and let Θ, Θ_1 and Θ_2 be sets of solutions such that $\Theta = \text{merge}(\Theta_1, \Theta_2, s, f)$, and let the optimal solutions from these sets be given by $\theta = \text{opt}(\Theta, s)$, $\theta_1 = \text{opt}(\Theta_1, s_1)$ and $\theta_2 = \text{opt}(\Theta_2, s_2)$. Then, according to Eq. (4.24) an optimization task is splittable if and only if:

$$\theta = \text{opt}(\text{merge}(\theta_1, \theta_2, s, f), s) \tag{4.25}$$

For this to hold the solution set θ must be both a subset and a superset of the right-hand side of Eq. (4.25). We consider both next. However, for brevity of notation, we leave out the states s, s_1, s_2 and the branching feature f.

1. θ is a subset:

$$\theta \subseteq \text{opt}(\text{merge}(\theta_1, \theta_2)) \tag{4.26}$$

   By definition, any $v \in \theta = \text{opt}(\Theta)$ must be feasible and not dominated by $\text{merge}(\Theta_1, \Theta_2)$, which implies that it is also not dominated by its subset $\text{merge}(\theta_1, \theta_2)$, and therefore Eq. (4.26) holds.

2. θ is a superset:

$$\theta \supseteq \text{opt}(\text{merge}(\theta_1, \theta_2)) \tag{4.27}$$

   By contradiction, assume that Eq. (4.27) is false. This means there must exists a solution $v = v_1 \oplus v_2$, with $v \in \text{opt}(\text{merge}(\theta_1, \theta_2))$, such that $v \notin \theta$. This means that v is either filtered out of θ by $\text{feas}(\Theta)$ or by $\text{nondom}(\Theta)$.

   (a) If v is filtered out by feas, it cannot be a member of $\text{opt}(\text{merge}(\theta_1, \theta_2))$, since it also applies feas.

   (b) If v is dominated by another solution in Θ, then there must exist a solution $v' \in \Theta : v' \succ v$, but $v' \notin \text{opt}(\text{merge}(\theta_1, \theta_2))$. Because of transitivity of $\succ$, if v is dominated by some $v' \in \Theta$, then also v must be dominated by some $v'' \in \theta$. Thus $v'' \succ v$ and because of Eq. (4.26), this $v'' \in \text{opt}(\text{merge}(\theta_1, \theta_2))$: a contradiction.

   In either case, the assumption is contradicted, and therefore Eq. (4.27) is true.

Since both Eq. (4.26) and Eq. (4.27) are true, Eq. (4.25) must be true and thus o is splittable.

Since both Prop. 4.4.3 and Theorem 4.4.6 share the conditions that the cost and transition function must be Markovian, and because o's cost and transition function are Markovian and o is splittable, optimization task o is separable by Def. 4.4.2. Since this holds for any optimization task o that satisfies the conditions of Theorem 4.4.6, these conditions are sufficient to prove separability.

The conditions are also necessary. The condition of a Markovian cost and transition function is shared among Prop. 4.4.3 and Theorem 4.4.6 and therefore obviously necessary. Therefore, what is left to show is that order preservation and anti-monotonic constraints are necessary conditions to show the principle of optimality. For brevity of notation, we again leave out the states s, s_1, s_2 and the branching feature f.

Order preservation Let $\Theta = \text{merge}(\Theta_1, \Theta_2)$, with $\theta = \text{opt}(\Theta), \theta_1 = \text{opt}(\Theta_1)$ and $\theta_2 = \text{opt}(\Theta_2)$. Now assume that the splitting property holds: This means that for all $v = v_1 \oplus v_2$, with $v \in \theta$, also $v_1 \in \theta_1$ and $v_2 \in \theta_2$ and the other way around. Now assume that order preservation is not necessary. This means $\exists v_1' \notin \theta_1$, with $v' = v_1' \oplus v_2$. Consider two cases: if $v' \in \theta$, then $v_1' \in \theta_1$: a contradiction. However, if $v' \notin \theta$, then also $v \notin \theta$: a contradiction.

Anti-monotonicity If the constraint c is not anti-monotonic, then there exists a solution $v = v_1 \oplus v_2$ with $\neg c(v_1)$, but with $v \in \text{opt}(\Theta)$. If $\neg c(v_1)$, then $v_1 \notin \text{opt}(\Theta_1)$. Thus, the splitting property is not satisfied.

Since each condition separately is necessary and all conditions together are sufficient, Theorem 4.4.6 holds. □

4

4.A.3. Proof of Theorem 4.4.7 about the Optimality of STreeD

Theorem 4.4.7 states that any *separable* optimization task can be solved to optimality using Eq. (4.8) from the main text. Here, we provide a proof for this theorem.

Proof. Let $\Theta(s, d)$ denote the set of all possible solution values to a decision tree search problem with state s, and d the maximum depth of the tree, and let $\theta(s, d)$ be the set of nondominated and feasible solutions for a given separable optimization task $o = \langle g, t, \succ, \oplus, c, s_0 \rangle$:

$$\theta(s, d) = \text{opt}(\Theta(s, d), s) \tag{4.28}$$

We need to prove that for every state s and depth d, Eq. (4.8) returns the optimal solution:

$$\theta(s, d) = T(s, d) \tag{4.29}$$

We will prove this by induction over d.

Base step: If $d = 0$, because the cost function g is Markovian, the solution set $\Theta(s, 0)$ can be described as:

$$\Theta(s, 0) = \bigcup_{\hat{k} \in \mathcal{K}} \left\{ g(s, \hat{k}) \right\} \tag{4.30}$$

Therefore the optimal solution set as defined by Eq. (4.28) is precisely what Eq. (4.8) returns for $d = 0$:

$$\theta(s, 0) = T(s, 0) \tag{4.31}$$

Therefore, since we made no further assumptions, the base step of this proof holds.

Induction step: For $d > 0$, we assume that Eq. (4.8) returns the optimal solution for $d - 1$:

$$\theta(s, d-1) = T(s, d-1) \tag{4.32}$$

We now need to show that Eq. (4.8) when $d > 0$ returns the optimal solution:

$$\theta(s, d) = \text{opt}(\Theta(s, d)) = \text{opt}\left(\bigcup_{f \in \mathcal{F}} \text{merge}\left(T(t(s, f), d-1), T(t(s, \bar{f}), d-1) \right) \right) \tag{4.33}$$

Define $\Theta_f(s,d)$ as denoting the set of all possible solution values with feature f as the branching feature of the root:

$$\Theta_f(s,d) = \text{merge}\left(\Theta(t(s,f),d-1), \Theta(t(s,\bar{f}),d-1), s, f\right) \tag{4.34}$$

When $d > 0$, the set $\Theta(s,d)$ can be described as the combination of all sets :

$$\Theta(s,d) = \bigcup_{f\in\mathcal{F}} \Theta_f(s,d) \tag{4.35}$$

Any solution that would be filtered out in a subset would also be filtered out in the whole set, therefore:

$$\text{opt}(\cup_{\Theta'}\Theta') = \text{opt}(\cup_{\Theta'}\text{opt}(\Theta')) \tag{4.36}$$

We can now derive the following:

$$\begin{aligned}\theta(s,d) &= \text{opt}(\Theta(s,d)) \\ &= \text{opt}\left(\cup_{f\in\mathcal{F}}\Theta_f(s,d)\right) \\ &= \text{opt}\left(\cup_{f\in\mathcal{F}}\text{opt}\left(\Theta_f(s,d)\right)\right)\end{aligned} \tag{4.37}$$

Because of Eq. (4.37), in order to prove Eq. (4.33), we only need to show the following:

$$\theta_f(s,d) = \text{opt}\left(\text{merge}\left(T(t(s,f),d-1), T(t(s,\bar{f}),d-1), s, f\right), s\right) \tag{4.38}$$

Since the induction step assumes T computes optimal trees for $d-1$, we may here substitute with Eq. (4.32):

$$\theta_f(s,d) = \text{opt}\left(\text{merge}\left(\theta(t(s,f),d-1), \theta(t(s,\bar{f}),d-1), s, f\right), s\right) \tag{4.39}$$

Because the optimization task is separable, it is splittable, and therefore Eq. (4.38) holds.

Therefore the induction step also holds. Since this proof is shown without any further assumptions, by mathematical induction Theorem 4.4.7 is true. □

4.A.4. Threshold Constraints

Many constraints, such as capacity constraints, can be considered as an objective with a threshold value β:

$$c_\beta(v,s) = \mathbb{1}(v \succ \beta) \tag{4.40}$$

Eq. (4.40) means that a solution is feasible if its solution value is better ($\succ$) than the threshold. E.g., when minimizing, any solution with a value lower than the bound is feasible.

Constraints need to be anti-monotonic to be separable. Threshold constraints are anti-monotonic provided that the combining operator $\oplus$ is worsening:

Definition 4.A.2. (Worsening operator) A combining operator $\oplus$ is called *worsening* if and only if $v = v_1 \oplus v_2 \rightarrow v_1 \succeq v \wedge v_2 \succeq v$.

For example, when setting prices to maximize revenue under a maximum supply constraint, you can calculate the expected demand in each node, sum demand from several nodes by using addition, and then prune trees that exceed this maximum supply by using a threshold constraint.

Proposition 4.A.3. *Given an optimization task* $o = \langle g, t, \succ, \oplus, c_\beta, s_0 \rangle$*, if the combining operator* $\oplus$ *satisfies the* worsening *property, then the threshold constraint* c_β *is* anti-monotonic*.*

Proof. Given any $v_1 \in \Theta_1, v_2 \in \Theta_2$ with $\beta \succ v_1$ (or w.l.o.g., $\beta \succ v_2$), such that $v_1 \notin \theta_1$. Let $v = v_1 \oplus v_2$ with $\oplus$ worsening, then $\beta \succ v_1 \succeq v$ and therefore $v \notin \theta$, and therefore c_β is anti-monotonic. □

4

4.A.5. Combining Multiple Optimization Tasks

We now show that a combination of separable optimization tasks is also separable. This is important for multi-objective optimization or for optimizing with a constraint. For example, we can maximize revenue in the first task, while respecting a maximum supply constraint in the second task. Let $o_a = \langle g_a, t_a, \succ_a, \oplus_a, c_a, s_0^a \rangle$ and $o_b = \langle g_b, t_b, \succ_b, \oplus_b, c_b, s_0^b \rangle$ be two separable optimization tasks. Two (or more) of such tasks can now be combined with the following equations:

$$g(\langle s_a, s_b \rangle, \hat{k}) = \langle g_a(s_a, \hat{k}), g_b(s_b, \hat{k}) \rangle \tag{4.41}$$

$$g(\langle s_a, s_b \rangle, f) = \langle g_a(s_a, f), g_b(s_b, f) \rangle \tag{4.42}$$

$$t(\langle s_a, s_b \rangle, f) = \langle t_a(s_a, f), t_b(s_b, f) \rangle \tag{4.43}$$

$$\langle v_1^a, v_1^b \rangle \succ \langle v_2^a, v_2^b \rangle \text{ iff } \langle v_1^a, v_1^b \rangle \neq \langle v_2^a, v_2^b \rangle \wedge \ v_1^a \succeq v_2^a \wedge v_1^b \succeq v_2^b \tag{4.44}$$

$$\langle v_1^a, v_1^b \rangle \oplus \langle v_2^a, v_2^b \rangle = \langle v_1^a \oplus_a v_2^a, v_1^b \oplus_b v_2^b \rangle \tag{4.45}$$

$$c(\langle v_a, v_b \rangle, \langle s_a, s_b \rangle) = \mathbb{1}(c_a(v_a, s_a) \wedge c_b(v_b, s_b)) \tag{4.46}$$

$$s_0 = \langle s_0^a, s_0^b \rangle \tag{4.47}$$

Eqs. (4.41)-(4.42) define the cost function as returning a tuple of the values that g_a and g_b return. Eq. (4.43) returns the result of both transition functions. Eq. (4.44) gives a partially defined comparison operator which states that a solution only dominates another solution if the solution values for both optimization tasks dominate both values of another solution. Eq. (4.45) defines the combination operator by applying the combination operator of the sub-tasks to the corresponding values of the left and right-hand side. Eq. (4.46) defines a new constraint that is only feasible when the solution is feasible according to both sub-constraints. Finally, Eq. (4.47) sets the starting state as the combination of both starting states.

Proposition 4.A.4. *Let* $o_a = \langle g_a, t_a, \succ_a, \oplus_a, c_a, s_0^a \rangle$ *and* $o_b = \langle g_b, t_b, \succ_b, \oplus_b, c_b, s_0^b \rangle$ *be two separable optimization tasks. Then the combined optimization task* $\langle g, t, \succ, \oplus, c, s_0 \rangle$ *defined by Eqs.* (4.41)-(4.47) *is also separable.*

To prove Prop. 4.A.4 it is necessary to show that the optimization task resulting from applying Eqs. (4.41)-(4.47) satisfies all the properties of being a separable optimization task.

Proof. Given two separable optimization tasks $o_a = \langle g_a, t_a, \succ_a, \oplus_a, c_a, s_0^a \rangle$ and $o_b = \langle g_b, t_b, \succ_b, \oplus_b, c_b, s_0^b \rangle$, and an optimization task $o = \langle g, t, \succ, \oplus, c, s_0 \rangle$ which is the result of applying Eqs. (4.41)-(4.47) to o_a and o_b.

Markovian: If g_a, g_b, t_a and t_b are Markovian, then g and t are also Markovian since their value can be fully expressed by calling the functions of the subtasks, which are Markovian, without considering any other decision variables.

Order preservation: If $\oplus_a$ preserves order over $\succ_a$ and $\oplus_b$ preserves order over $\succ_b$, then $\oplus$ also preserves order over $\succ$, because of the following.

Consider solutions $\langle v_1^a, v_1^b \rangle \in \text{opt}\,(\Theta_1, s_1)$, $\langle v_1^{a'}, v_1^{b'} \rangle \in \Theta_1$, so that $\langle v_1^a, v_1^b \rangle \succ \langle v_1^{a'}, v_1^{b'} \rangle$, consider $\langle v_2^a, v_2^b \rangle \in \text{opt}\,(\Theta_2, s_2)$ and assume that $\oplus$ is commutative. Then, because of Eq. (4.44), there are two options: either $v_1^a \succ_a v_1^{a'} \wedge v_1^b \succeq_b v_1^{b'}$, or $v_1^a \succeq_a v_1^{a'} \wedge v_1^b \succ_b v_1^{b'}$. W.l.o.g., choose the first. Since $\oplus_a$ preserves order over $\succ_a$ and $\oplus_b$ preserves order over $\succ_b$, we can now say that $v_1^a \oplus_a v_2^a \succ_a v_1^{a'} \oplus_a v_2^a$ and also $v_1^b \oplus_b v_2^b \succeq_b v_1^{b'} \oplus_b v_2^b$. Because of Eqs. (4.44)-(4.45), it follows:

$$\langle v_1^a, v_1^b \rangle \oplus \langle v_2^a, v_2^b \rangle \succ \langle v_1^{a'}, v_1^{b'} \rangle \oplus \langle v_2^a, v_2^b \rangle \tag{4.48}$$

Therefore $\oplus$ also preserves order over $\succ$.

If $\oplus$ is not commutative, the proof above should be repeated with the operands for every use of $\oplus$ switched around.

Anti-monotonicity: If c_a and c_b are both anti-monotonic, then for any given solution $v_1 = \langle v_1^a, v_1^b \rangle$ and v_2, and $v = \langle v^a, v^b \rangle = v_1 \oplus v_2$, and any given state s and branching decision f, such that $s_1 = t(s, f)$ and $s_2 = t(s, \bar{f})$: if $\neg c_a(v_1^a, s_1)$ (or similarly for c_b and v_2 and s_2), then also $\neg c_a(v^a, s)$, and therefore $\neg c(v, s)$, thus c is also anti-monotonic.

Conclusion Since the combined optimization task o satisfies all necessary properties, o is also separable. □

4.B. Pseudo-Code and Implementation

This section gives further details on the implementation of STreeD. First, we explain how upper and lower bounding works. Second, we expand on how to search for sparse trees and how to tune the number of nodes. Third, we provide conditions for caching. Fourth, pseudo-code for STreeD is given. Finally, a special solver for trees of depth two is discussed.

Upper bounds A set of solutions ub is an upper bound to another set Θ if for all solution v in Θ no solution in ub exists that is equal or dominates it:

$$\forall v \in \Theta : \neg \exists v' \in \text{ub} : v' \succeq v \tag{4.49}$$

An upper bound can be used to prune solutions that are dominated by it. The upper bound ub dominates a solution v if there exists a solution $v' \in \text{ub}$ that is equal to or dominates v:

$$\text{ub} \leq v \Leftrightarrow \exists v' \in \text{ub} : v' \succeq v \tag{4.50}$$

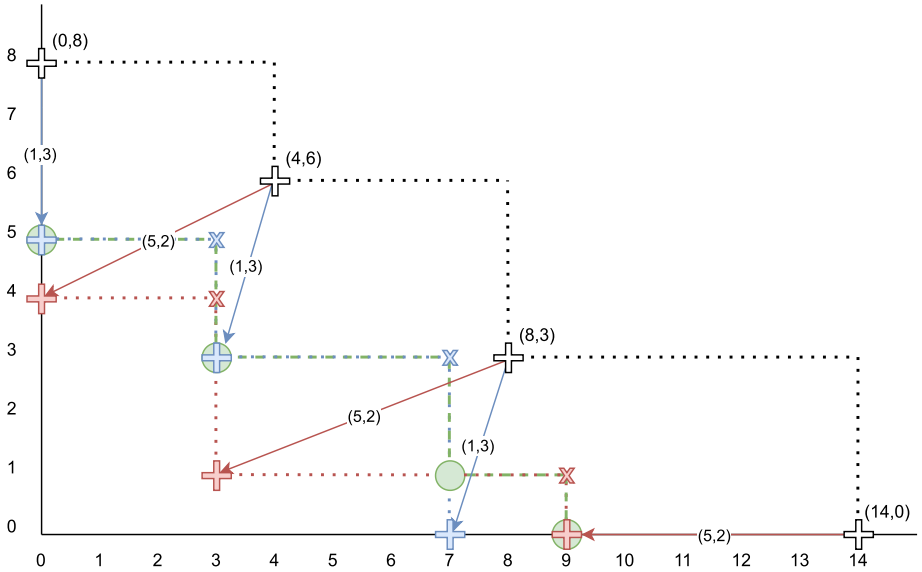


Figure 4.6: The result of subtracting the left solutions $\{(1,3),(5,2)\}$ from the upper bound $\{(0,8),(4,6),(8,3),(14,0)\}$. Subtracting the solution $(1,3)$ results in the blue Pareto front $\{(0,5),(3,3),(7,0)\}$. Subtracting the solution $(5,2)$ results in the red Pareto front $\{(0,4),(3,1),(9,0)\}$. Combining these two Pareto fronts results in the green Pareto front $\{(0,5),(3,3),(7,1),(9,0)\}$.

This upper bound is used to prune poor solutions earlier in the search. To signify this we introduce the following new notation:

$$\text{opt}(\Theta, s, \text{ub}) = \text{opt}\left(\{v \in \Theta \mid \text{ub} \not\leq v\}, s\right) \tag{4.51}$$

Subtracting In a branching node, when solutions Θ_1 are known for one subtree, these can be used to update the upper bounds for the other subtree, provided that an appropriate *subtraction* operator $\ominus$ exists. For totally ordered optimization tasks, the upper bound for the other subtree can be computed by subtracting the solution from the current upper bound. However, for partially ordered objectives, the procedure is more complicated. We here provide the method for element-wise additive objectives and leave the approach for more general objectives as future work.

Consider the example shown in Fig. 4.6 where the goal is to minimize both dimensions, and the subtraction $\ominus$ is defined as element-wise subtraction (but values below zero are set to zero). The idea is to find first for each solution $v \in \Theta_1$ the upper bound ub_v when v is subtracted from the upper bound ub (the blue and red plusses in Fig. 4.6).

$$\text{ub}_v = \text{nondom}(\{u \ominus v \mid u \in \text{ub}\}) \tag{4.52}$$

The final new upper bound ub_2 is the Pareto front that covers precisely the outer edge of all upper bounds ub_v, such that if any solution is dominated by all ub_v, it is dominated by ub_2.

To compute this new upper bound, we need to define some more functions and sets. Let $\prec$ decide if one solution is *reverse Pareto dominated*. Similarly, let $\overline{\text{nondom}}$ be the set of *reverse nondominated* solutions. Let the operator $\triangle$ combine two solutions v_1 and v_2 such that $v_1 \triangle v_2 \preceq v_1$ and also $v_1 \triangle v_2 \preceq v_2$. Similarly, let the operator $\triangledown$ combine two solutions v_1 and v_2 such that $v_1 \triangledown v_2 \succeq v_1$ and also $v_1 \triangledown v_2 \succeq v_2$. E.g., $(0, 5) \triangle (3, 3) = (3, 5)$. Finally, let the set $\mathcal{E}$ be the set of *extreme points*, with one extreme point per dimension with the value for that dimension set to the worst case.

For the example domains considered in this chapter, all of these operators ($\ominus$, $\prec$, $\triangle$, and $\triangledown$) can be defined. If $\oplus$ is (element-wise) addition, then $\ominus$ is (element-wise) subtraction, and $\prec$ is defined like $\succ$ but with the comparison reversed. In this case, $\triangle$ and $\triangledown$ are defined as (element-wise) max and min respectively. The extreme points for the non-linear metrics, for example, are $\mathcal{E} = \{(|\mathcal{D}^+|, 0), (0, |\mathcal{D}^-|)\}$.

With help of the functions above, one can compute the *reverse Pareto front* of ub_v (the blue and red crosses in Fig. 4.6).

$$\text{reverse}(\Theta) = \text{nondom}(\{v_1 \triangle v_2 \mid v_1, v_2 \in \Theta \cup \mathcal{E}, v_1 \neq v_2\}) \tag{4.53}$$

And similarly, to reverse back:

$$\overline{\text{reverse}}(\Theta) = \overline{\text{nondom}}(\{v_1 \triangledown v_2 \mid v_1, v_2 \in \Theta, v_1 \neq v_2\}) \tag{4.54}$$

The upper bounds ub_v can be combined using $\overline{\text{nondom}}$:

$$\text{ub}_2^* = \overline{\text{nondom}}\left(\bigcup_{v \in \Theta_1} \text{ub}_v\right) \tag{4.55}$$

However, the set ub_2^* is not always the best upper bound. E.g., in Fig. 4.6 it would not include $(7, 1)$, and therefore would not consider the solution $(8, 2)$ dominated, even though it is.

The quality of the upper bound can be improved by also considering the reverse Pareto fronts. The reverse Pareto fronts of the solutions $v \in \Theta$ can be combined using $\overline{\text{nondom}}$ and then reversed back with $\overline{\text{reverse}}$ as follows:

$$\text{ub}_2' = \overline{\text{reverse}}\left(\overline{\text{nondom}}\left(\bigcup_{v \in \Theta_1} \text{reverse}(\text{ub}_v)\right)\right) \tag{4.56}$$

The final upper bound for the other subtree (the green circles in Fig. 4.6) is obtained by combining the two sets with $\overline{\text{nondom}}$:

$$\text{ub}_2 = \text{ub} - \Theta_1 = \overline{\text{nondom}}(\text{ub}_2^* \cup \text{ub}_2') \tag{4.57}$$

Lower bounds The set lb is a lower bound to Θ, if for all solutions v in Θ there is no solution in lb which is dominated by v:

$$\forall v \in \Theta : \neg \exists v' \in \text{lb} : v \succ v' \tag{4.58}$$

We compute lower bounds similarly as Demirović and Stuckey (2021). We use the notation $\text{LB}(s, d)$ for retrieving a lower bound from the cache or by computing it by using a similarity bound for a state $s = \langle \mathcal{D}, F \rangle$. Any optimal solution in the cache is also a lower bound. Furthermore, if a tree search node is infeasible (no feasible solution below the upper bound ub exists), then the upper bound for this node becomes a lower bound that is stored in the cache.

The lower bound may also be computed by using a *similarity bound*, but this only applies when the optimization task

1. has a subtraction operator $\ominus$;
2. does not have constraints;
3. is *context independent*; and
4. and the optimization task is *per-instance* (element-wise) additive.

Definition 4.B.1 (Context independent)**.** An optimization task is *context independent* if and only if the cost function does not depend on any of the parent nodes' branching decisions.

For example, cost-sensitive classification with discounts, as introduced in this chapter, is not context independent.

Definition 4.B.2 (Per-instance (element-wise) additive)**.** An optimization task is *per-instance (element-wise) additive*, if the cost function for label assignments can be written as the sum of the costs from a per-instance cost function h:

$$g(\langle \mathcal{D}, F \rangle, \hat{k}) = \sum_{(x,k) \in \mathcal{D}} h(x, k, \hat{k}) \tag{4.59}$$

If all these four conditions are met, the similarity bound can be used. Its definition is generalized from the definition given in (Demirović and Stuckey, 2021). Instead of subtracting the misclassification score per class from an existing solution, the worst-case value for instances of that class is subtracted by use of $\ominus$. Let w_k be the worst case contribution of a single instance with class k to the objective, and let n_k be the number of instances in a dataset $\mathcal{D}$ that are not part of a dataset for a cached solution $\mathcal{D}^c$: i.e. if $\mathcal{D}_k = \{(x, k') \in \mathcal{D} \mid k' = k\}$, then $n_k = |D_k \setminus D_k^c|$. Now assume each of these n_k instances is misclassified and therefore w_k is added to the objective for each of these instances, then a lower bound can be computed from a cached solution Θ^c as follows:

$$\text{lb}(\mathcal{D}) = \{v \ominus \sum_{k \in \mathcal{K}} n_k w_k \mid v \in \Theta^c\} \tag{4.60}$$

In this equation, the sum operator uses $\oplus$ to sum.

Once the lower bound for a search node is computed, it can be compared to the upper bound ub to check if any possible improvement exists, and if not, the node is

pruned from the tree search. This happens whenever lb > ub, i.e., every element of the upper bound is reverse dominated by at least one element of the lower bound:

$$\text{lb} > \text{ub} \Leftrightarrow \forall v \in \text{ub} : \exists v' \in \text{lb} : v' \prec v \tag{4.61}$$

Computing these upper and lower bounds can be quite costly since it involves a product operation of two sets. To reduce the computation time for applying the subtraction or to combine two lower bounds for two subtrees, we apply a new procedure in which we first reduce the two sets to a representative subset using $\triangle$ in such a way that the resulting bounds are still valid. These representative subsets are then combined to get a new valid upper or lower bound. This decreases the computation time of the bound but prunes slightly fewer nodes.

Sparse trees Optimal decision trees may overfit on the training data (Dietterich, 1995) and therefore typically a sparsity coefficient is introduced that adds a cost for every branching node, as for example done in (Bertsimas and Dunn, 2017; Hu et al., 2019; Lin et al., 2020). In the work presented here, we do not consider such a sparsity coefficient, although it could trivially be added. Instead, STreeD optimizes directly for a given maximum number of nodes. This number of nodes can then be tuned (as explained below). In the DP formulation in the main text, this is left out for brevity. For completeness, the DP formulation should be changed by considering the following equation in a branching node ($d > 0$), as is similarly proposed in (Demirović et al., 2022):

$$\text{opt}\left(\bigcup_{f\in\mathcal{F},i\in[0,n-1]} \text{merge}\left(T(t(s,f),d-1,i),T(t(s,\bar{f}),d-1,n-i-1),s,f\right)\right) \tag{4.62}$$

With this addition, STreeD can be tuned to optimize the number of nodes to prevent overfitting. This is done by using part of the training data as validation data and train STreeD on the rest of the training data for an increasing number of nodes. This is repeated five times and the number of nodes that yields on average the best result is used as the final maximum number of nodes.

Caching STreeD supports two forms of caching: branch caching and dataset caching. Dataset caching is typically slightly faster than branch caching (Demirović et al., 2022). However, dataset caching cannot be used for context-dependent optimization tasks, because dataset similarity is no longer sufficient to identify similar subproblems. Similarly, for some optimization tasks, branch caching should be replaced by equivalent state caching, which we leave as future work. In this chapter, dataset caching is used, except for cost-sensitive classification and group fairness, which are context dependent and use branch caching.

Pseudo-code Pseudo-code for STreeD is given in Algorithm 6. If a leaf node is reached, the cost function is used to compute optimal solutions. The next two checks in the pseudo-code check if the maximum number of nodes and the maximum depth are compatible. If they are, the cache is checked for optimal solutions or a lower bound. The search is stopped if the cached lower bound is dominated by the current upper bound. If the cached solution is optimal, it is returned.

Algorithm 6: Tree search of depth d for an optimization task $\langle g, t, \succ, \oplus, c, s_0 \rangle$ for state s, for a feature set $\mathcal{F}$ and a maximum number of nodes n.

$T(s, d, n, \text{ub})$
  if $d = 0 \vee n = 0$ **then**
    return opt $\left(\cup_{\hat{k} \in \mathcal{K}} g(s, \hat{k}), s, \text{ub}\right)$
  if $n > 2^d - 1$ **then**
    return $T(s, d, 2^d - 1, \text{ub})$
  if $d > n$ **then**
    return $T(s, d, d, \text{ub})$
  $\langle \Theta, \text{lb}, \text{stat} \rangle \leftarrow \text{cache}[s, d, n]$
  if lb $>$ ub **then return** $\emptyset$
  if stat $=$ optimal **then return** opt (Θ, s, ub)
  $\Theta \leftarrow \emptyset$
  for $f \in \mathcal{F}, n_L \in [0, n-1]$ **do**
    $n_R \leftarrow n - n_L - 1$
    $\text{lb}_L \leftarrow \text{LB}(t(s, \bar{f}), d-1, n_L)$
    $\text{lb}_R \leftarrow \text{LB}(t(s, f), d-1, n_R)$
    $\text{lb} \leftarrow \text{merge}(\text{lb}_L, \text{lb}_R, s, f)$
    if lb $>$ ub **then continue**
    $\text{ub}_L = \text{ub} \ominus g(s, f)$
    $\Theta_L \leftarrow T(t(s, \bar{f}), d-1, n_L, \text{ub}_L)$
    if $\Theta_L = \emptyset$ **then continue**
    $\text{ub}_R = \text{ub} \ominus \Theta_L \ominus g(s, f)$
    $\Theta_R \leftarrow T(t(s, f), d-1, n_R, \text{ub}_R)$
    if $\Theta_R = \emptyset$ **then continue**
    $\Theta_{\text{new}} \leftarrow \text{opt}(\text{merge}(\Theta_L, \Theta_R, s, f), s)$
    $\text{ub} \leftarrow \text{opt}(\text{ub} \cup \Theta_{\text{new}}, s)$
    $\Theta \leftarrow \text{opt}(\Theta \cup \Theta_{\text{new}}, s, \text{ub})$
  if $\Theta = \emptyset$ **then**
    $\text{cache}[s, d, n] \leftarrow \langle \emptyset, \text{ub}, \text{lower bound} \rangle$
    return $\emptyset$
  $\text{cache}[s, d, n] \leftarrow \langle \Theta, \Theta, \text{optimal} \rangle$
  return Θ

Then, for every possible branching feature and every possible node division, the algorithm does the following. It computes lower bounds for the left and right subtree, merges these lower bounds, and skips this feature if the upper bound dominates the lower bound. Then a recursive call is performed to compute optimal solutions to the left subtree. The upper bound for the left tree is updated by subtracting the branching costs. Solutions from this recursive call are used to update the upper bound for the right subtree and the right subtree is also solved. Solutions from both subtrees are merged into a new set of solutions which are used to update the current

set of solutions and the upper bound.

When all branching features have been checked and no feasible solution is found (better than the upper bound), the current upper bound is stored as a lower bound. If feasible solutions have been found, the current set of solutions is stored in the cache and the solutions are returned.

In case the comparison operator $\succ$ is fully defined (a *total ordering* of all solutions exists), such as is the case with minimizing misclassification score, then the nondom operation always returns a set of precisely one solution (or zero if infeasible). In this case, nondom can be replaced by min (as defined by $\succ$), simplifying the implementation and computation.

Depth-two solver Demirović et al. (2022) have shown that the runtime of optimal decision tree search can be improved considerably by use of a special solver for trees of maximum depth two. The core idea is to pre-compute class frequencies by looping over only the *present features* for every instance in a dataset. We generalize this concept in STreeD and explain it below.

STreeD's use of the depth-two solver is restricted under the following conditions:

1. the cost of a leaf node can be expressed as a function over the contributions of individual instances;
2. the cost of a leaf node is *context independent* (see Def. 4.B.1, note that the branching costs may be context dependent); and
3. the branching costs are equal for datasets with the same size, i.e., $|\mathcal{D}_1| = |\mathcal{D}_2| \rightarrow g(\langle \mathcal{D}_1, F\rangle, f) = g(\langle \mathcal{D}_2, F\rangle, f)$ for all $f \in \mathcal{F} \times \{0, 1\}$.

For all optimization tasks considered in this chapter, these conditions are met.

To use the depth-two solver, the following functions and values need to be defined:

1. a 'depth-two' solution space $\mathcal{W}$;
2. an 'empty' initial solution value $w_0 \in \mathcal{W}$ (typically zero), the solution value of a leaf with zero instances;
3. a per-instance cost function $j : \mathcal{X} \times \mathcal{K} \times \mathcal{K} \rightarrow \mathcal{W}$ such that $j(x, k, \hat{k})$ returns the 'depth-two' costs of instance (x, k) when assigned label $\hat{k}$;
4. a 'depth-two' combining operator $\oplus' : \mathcal{W} \times \mathcal{W} \rightarrow \mathcal{W}$ that combines two solution values into one;
5. a 'depth-two' subtract operator $\ominus' : \mathcal{W} \times \mathcal{W} \rightarrow \mathcal{W}$, such that $(w_1 \oplus' w_2) \ominus' w_2 = w_1$ for all $w_1, w_2 \in \mathcal{W}$;
6. a transformation function $q : \mathcal{W} \rightarrow \mathcal{V}$ that transforms a 'depth-two' solution value into a solution value in the original solution value space $\mathcal{V}$;
7. a branching cost function $r : \mathcal{S} \times \mathbb{N}^+ \times \mathcal{F} \times \{0, 1\} \rightarrow \mathcal{V}$, such that $r(s, n, f)$ returns the cost of branching on f in state s with $n = |\mathcal{D}|$.

In this chapter, for all optimization tasks $\mathcal{W} = \mathcal{V}$, $\oplus' = \oplus$, and $j(x, k, \hat{k}) = g(\langle\{(x, k)\}, \emptyset\rangle, \hat{k})$, but this is not necessarily true for future optimization tasks.

The function j provides a per-instance breakdown of the costs which are summed with the $\oplus'$ operator. These are used to precompute the costs of all possible leaf nodes. Let $n(f_i)$ and $n(f_i, f_j)$ be the size of $\mathcal{D}_{f_i}$ and $\mathcal{D}_{f_i,f_j}$ respectively. Similarly, let $w(\hat{k})$ be the cost of assigning all instances in $\mathcal{D}$ label $\hat{k}$, and let $w(\hat{k}, f_i)$ and $w(\hat{k}, f_i, f_j)$ be the cost of assigning all instances in $\mathcal{D}_{f_i}$ and $\mathcal{D}_{f_i,f_j}$ respectively. By

generalizing the result from Demirović et al. (2022), the following statements hold:

$$n(\bar{f}_i) = |\mathcal{D}| - n(f_i) \tag{4.63}$$
$$n(f_i, \bar{f}_j) = n(f_i) - n(f_i, f_j) \tag{4.64}$$
$$n(\bar{f}_i, f_j) = n(f_j) - n(f_i, f_j) \tag{4.65}$$
$$n(\bar{f}_i, \bar{f}_j) = |\mathcal{D}| - n(f_i) - n(f_j) + n(f_i, f_j) \tag{4.66}$$
$$w(\hat{k}, \bar{f}_i) = w(\hat{k}) \ominus' w(\hat{k}, f_i) \tag{4.67}$$
$$w(\hat{k}, f_i, \bar{f}_j) = w(\hat{k}, f_i) \ominus' w(\hat{k}, f_i, f_j) \tag{4.68}$$
$$w(\hat{k}, \bar{f}_i, f_j) = w(\hat{k}, f_j) \ominus' w(\hat{k}, f_i, f_j) \tag{4.69}$$
$$w(\hat{k}, \bar{f}_i, \bar{f}_j) = w(\hat{k}) \ominus' w(\hat{k}, f_i) \ominus' w(\hat{k}, f_j) \oplus' w(\hat{k}, f_i, f_j) \tag{4.70}$$

Furthermore, let $f \in x$ denote that feature f is satisfied by x. The depth-two solver can now be generalized as shown in Algorithm 7. It first precomputes the dataset sizes and costs, then finds the best left and right solutions for each possible split, and finally returns the best solution.

4.C. Related Work for the Application Domains

In this appendix, we provide additional related work for four of the learning tasks considered in this work: cost-sensitive classification, prescriptive policy generation, nonlinear classification metrics, and group fairness.

4.C.1. Cost-Sensitive Classification

Lomax and Vadera (2013) provide an overview of methods for cost-sensitive classification with decision trees, most of which use top-down induction with varying cost functions for deciding on the best split (Norton, 1989; Núñez, 1991; Tan, 1993; Ling et al., 2006). Each of these splitting criteria is mostly the same, except for a slight difference in the trade-off between accuracy and feature costs. This therefore is a good example of the drawback of top-down induction by splitting criteria: the objective is not directly translatable to a good splitting criterion. To address this issue Min and Zhu (2012) generalize a number of those methods using a hyperparameter that decides on the trade-off between information gain and feature costs. ICET (Turney, 1995) does the same but uses a genetic algorithm to decide the hyperparameter. All of these methods except ICET, however, do not consider misclassification costs, and even ICET only does so indirectly.

Lomax and Vadera (2017) co-optimize accuracy and costs by approaching the problem as a multi-armed bandit scenario. This means that in every branching node, they repeatedly try random splits (possibly multiple consecutive random splits) and choose the split which on average has the best expected costs based on this random exploration.

Maliah and Shani (2021) formulate the problem as a Partially Observable Markov Decision Process (POMDP). However, they observe that the action and state space of their model is too big, and therefore they present a smaller MDP that approximates

Algorithm 7: Tree search of depth two using a special depth-two solver.

$T_{d2}(s, \langle D, F\rangle)$
  // Pre-compute dataset sizes
  $n(f_i) \leftarrow 0 \quad \forall f_i \in \mathcal{F}$
  $n(f_i, f_j) \leftarrow 0 \quad \forall f_i, f_j \in \mathcal{F}$ s.t. $i < j$
  for $(x, k) \in \mathcal{D}$ **do**
    for $f_i \in x$ **do**
      $n(f_i) \leftarrow n(f_i) + 1$
      for $f_j \in x$ **do**
        $n(f_i, f_j) \leftarrow n(f_i, f_j) + 1$
  // Pre-compute costs
  $w(\hat{k}) \leftarrow w_0 \quad \forall \hat{k} \in \mathcal{K}$
  $w(\hat{k}, f_i) \leftarrow w_0 \quad \forall \hat{k} \in \mathcal{K}, f_i \in \mathcal{F}$
  $w(\hat{k}, f_i, f_j) \leftarrow w_0 \quad \forall \hat{k} \in \mathcal{K}, f_i, f_j \in \mathcal{F}$ s.t. $i < j$
  for $(x, k) \in \mathcal{D}, \hat{k} \in \mathcal{K}$ **do**
    $w(\hat{k}) \leftarrow w(\hat{k}) \oplus' j(x, k, \hat{k})$
    for $f_i \in x$ **do**
      $w(\hat{k}, f_i) \leftarrow w(\hat{k}, f_i) \oplus' j(x, k, \hat{k})$
      for $f_j \in x$ **do**
        $w(\hat{k}, f_i, f_j) \leftarrow w(\hat{k}, f_i, f_j) \oplus' j(x, k, \hat{k})$
  // Find optimal subtrees
  for $f_i, f_j \in \mathcal{F}$ *s.t.* $f_i \neq f_j$ **do**
    for $\hat{k}_L, \hat{k}_R \in \mathcal{K}$ *s.t.* $\hat{k}_L \neq \hat{k}_R$ **do**
      $\text{cost}_L \leftarrow q(w(\hat{k}_L, \bar{f}_i, \bar{f}_j) \oplus q(w(\hat{k}_R, \bar{f}_i, f_j)$
      $\text{cost}_L \leftarrow \text{cost}_L \oplus r(t(s, \bar{f}_i), n(\bar{f}_i, \bar{f}_j), \bar{f}_j) \oplus r(t(s, \bar{f}_i), n(\bar{f}_i, f_j), f_j)$
      $\text{cost}_R \leftarrow q(w(\hat{k}_L, f_i, \bar{f}_j) \oplus q(w(\hat{k}_R, f_i, f_j)$
      $\text{cost}_R \leftarrow \text{cost}_R \oplus r(t(s, f_i), n(f_i, \bar{f}_j), \bar{f}_j) \oplus r(t(s, f_i), n(f_i, f_j), f_j)$
      if $\text{cost}_L \succ \text{best-cost}_L(f_i)$ **then**
        $\text{best-cost}_L(f_i) \leftarrow \text{cost}_L$
      if $\text{cost}_R \succ \text{best-cost}_R(f_i)$ **then**
        $\text{best-cost}_R(f_i) \leftarrow \text{cost}_R$
  return $\text{opt}\left(\bigcup_{f \in \mathcal{F}} \text{merge}(\text{best-cost}_L(f), \text{best-cost}_R(f), s, f)\right)$

the POMDP. For the MDP model, they generate trees for all possible subsets of features using a standard heuristic that ignores costs. The MDP's state space is the set of all leaf nodes of all those generated trees. The resulting MDP is acyclic, and therefore its value function can be computed in one bottom-up pass through the belief states.

As far as we know, Nijssen and Fromont (2010) and Zubek and Dietterich (2002) are the only ones to provide an optimal formulation. The latter do so by formulating it as an MDP. However, in their experiments, they decide to use a relaxed formulation

instead because of memory limitations.

4.C.2. Prescriptive Policy Generation

Athey and Imbens (2016) use trees to estimate the causal effect of treatments. They use part of the dataset to estimate heterogeneous partitions and the rest is used to estimate treatment effects in the leaves, including confidence intervals for these values. However, this method does not directly assign policies, and in the case of non-binary treatments, several trees need to be combined to produce a policy tree, reducing its interpretability.

Kallus (2017) combines the prediction and prescription into one learning problem and proposes both a top-down induction heuristic and an optimal MIP method that directly optimizes prescription error. Their methods work under the assumption that the training data is either produced in a randomized experiment or that the generated tree partitions the data such that the historical treatment assignment becomes independent of the individual. However, in practice, the optimal MIP method does not scale to sufficient depth to reach such a partition. Their results therefore also show that the optimal MIP method in most situations performs worse than other solution methods.

Bertsimas et al. (2019) improve on (Kallus, 2017) by observing that the performance of prescriptive trees relies not only on a good heterogeneous partition but also on the quality of the prediction of outcomes. Therefore they propose an optimal MIP formulation that co-optimizes both the prescription and prediction error in one objective. To circumvent the scalability issues for finding the global optimum, they repeatedly use a coordinate descent method (Bertsimas and Dunn, 2019) to find local optima and select the best tree.

Biggs et al. (2021) and Zhou et al. (2022) note the disadvantage of trying to incorporate the predictive aspect into the MIP model. Instead they use a separate predictive teacher model and use this model to guide a top-down greedy heuristic. Zhou et al. (2022) also provide an optimal recursive search method, but they did not incorporate any of the scalability improvements from similar approaches (Aglin et al., 2020a; Demirović et al., 2022) such as caching and lower-bounding.

Amram et al. (2022) take a similar approach, but instead of a greedy heuristic, they propose to adapt the MIP model of (Bertsimas et al., 2019) by updating the objective to incorporate the counterfactual data obtained from a predictive model, using objectives from causal inference. Their experiments show its advantage over both the greedy heuristics and the method by Bertsimas et al. (2019).

Jo et al. (2021) analytically show the limitations of both (Kallus, 2017) and (Bertsimas et al., 2019): their assumption is generally not satisfied in conditionally randomized experiments. Both methods can return trees that actively violate their own assumption. Therefore Jo et al. (2021), like Amram et al. (2022), propose to drop this assumption and use separate predictive and prescriptive models. Apart from this, they show that under mild conditions their method provides an optimal out-of-sample policy when the number of training samples approaches infinity. Their model also allows for the addition of budget constraints. They propose to use the same objectives as Amram et al. (2022), but their MIP formulation is based on the

more scalable formulation of Aghaei et al. (2024), so unlike Bertsimas et al. (2019) and Amram et al. (2022) they do not need to fall back on coordinate descent for local optima. However, despite their scalability improvements, almost half of the instances in their experiments are still not solvable within one hour.

Finally, Subramanian et al. (2022) find near-optimal solutions by using column generation to improve scalability. Furthermore, their model allows for the addition of a variety of constraints.

4.C.3. Nonlinear Classification Metrics

Lin et al. (2020) also propose a DP method for solving a variety of objectives, such as balanced accuracy and F1-score. However, they observe that F1-score is harder to optimize than other metrics that are monotonic in relation to the number of false positives and negatives because the best assignment in one leaf node is dependent on the assignment in another node. Therefore, they simplify the problem by introducing a parameter ω that helps determine which label should be assigned in leaf nodes, but as a result, the solution is no longer guaranteed to be optimal.

Xin et al. (2022) present a DP method for finding the full Rashomon set of sparse decision trees, i.e., the set of almost-optimal decision trees (including the optimal). They then show how a Rashomon set for trees optimized for accuracy can be used to find the Rashomon set for F1-score. For this, they provide a bound on the accuracy score such that an Accuracy Rashomon set covers the F1-score Rashomon set.

4.C.4. Group Fairness

Fairness in decision trees was first considered by Kamiran and Calders (2009) who proposed to preprocess the data to guide the decision tree training process to less discriminating trees. Later, Kamiran et al. (2010) proposed new splitting criteria to enable top-down induction using a mix of information gain in the outcome label and the sensitive feature. However, they concluded that this was not sufficient and instead proposed after-the-fact relabeling to get good results.

Fairness constraints were first considered in MIP models for optimal decision trees by Verwer and Zhang (2017). Aghaei et al. (2019) later developed a MIP model for optimal trees for either a group fairness or individual fairness constraint. Jo et al. (2023) propose a new MIP model based on a max-flow formulation that scales better than previous models.

These MIP models, however, still do not scale well beyond small datasets and small tree depths. Therefore Wang et al. (2022) propose to break down the problem and use MIP as a look-ahead procedure that decides on the best feature to split on, but does not solve the whole problem as a monolith. Their experiments show better scalability, but their method is no longer guaranteed to return optimal solutions.

Van der Linden et al. (2022), on the other hand, propose an optimal DP method that optimizes a global fairness constraint by using upper and lower bounds on the final discrimination value for subtrees to enable early comparison and pruning. Their results show order-of-magnitude improvements on runtime, and their method can handle much larger datasets than both the MIP methods and the iterative approach.

5

Survival Analysis

Preface This chapter applies the general framework of Chapter 4 to survival analysis. The objective of survival analysis is highly nonlinear, so hard to solve with mixed-integer linear programming, but we can still use dynamic progamming. To improve scalability, we show how the loss function can be broken down so that the depth-two subroutine presented in Chapter 4 can be used.

Abstract Survival analysis studies and predicts the time of death, or other singular unrepeated events, based on historical data, while the true time of death for some instances is unknown. Survival trees enable the discovery of complex nonlinear relations in a compact human comprehensible model, by recursively splitting the population and predicting a distinct survival distribution in each leaf node. We use dynamic programming to provide the first survival tree method with optimality guarantees, enabling the assessment of the optimality gap of heuristics. We improve the scalability of our method through a special algorithm for computing trees up to depth two. The experiments show that our method's runtime even outperforms some heuristics for realistic cases while obtaining similar out-of-sample performance with the state of the art.

5.1. Introduction

The aim of *survival analysis* is to study and predict the time until a singular unrepeated event occurs, such as death or mechanical failure. Applications include not only evaluating the effectiveness of medical treatment (Selvin, 2008), but also, for example, recidivism risk estimation in criminology (Chung et al., 1991), fish migration analysis (Castro-Santos and Haro, 2003) and studies on human migration and fertility (Eryurt and Koç, 2012).

Parts of this chapter have been published in Huisman, Van der Linden, and Demirović (2024), "Optimal Survival Trees: A Dynamic Programming Approach", in *Proceedings of AAAI-24*. This paper resulted from Tim Huisman's bachelor thesis project, supervised by Emir Demirović and me. Tim Huisman was responsible for most of the implementation and investigation, and part of the methodology. I did most of the conceptualization, methodology, and writing.

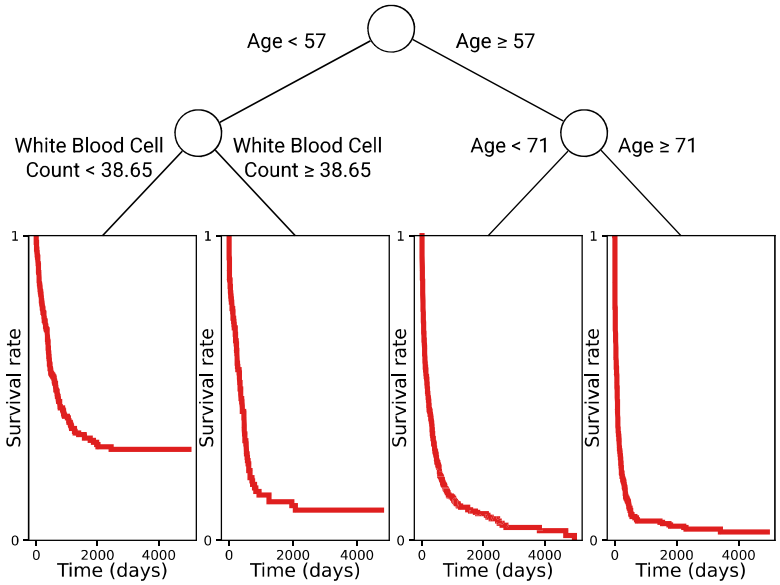


Figure 5.1: An example of a survival tree. Each leaf has a different survival distribution.

Survival analysis is challenging because the time of event of some instances is unknown, i.e., it is *censored*. This study focuses on the most common form of censoring: *right-censored* data, where the true time of the event is unknown, for example, because an instance survived beyond the end of the experiment record.

The application of survival analysis in the medical and other high-stakes domains motivates the use of human-interpretable machine learning models, such as *decision trees* (Rudin, 2019). In each of these leaf nodes, a survival curve can be computed, as shown, for example, in Fig. 5.1. Davis and Anderson (1989) provided one of the first survival tree methods, by applying recursive splitting of censored survival data with an interface similar to CART (Breiman et al., 1984). Other similar greedy top-down induction heuristics were developed by LeBlanc and Crowley (1993), Su and Fan (2004), and Hothorn et al. (2006), each applying different splitting techniques.

To improve the performance of survival trees, Bertsimas et al. (2022) recently proposed a local search method called Optimal Survival Trees (OST), based on the method proposed by Dunn (2018) and Bertsimas and Dunn (2019). Despite its name, OST does not provide a guarantee of global optimality, but iteratively finds local improvements to the tree structure and converges to a local optimum. To the best of our knowledge, there are no globally optimal decision tree methods for survival analysis yet.

Therefore, building on the promising result of optimizing a variety of ODT objectives with DP in Chapter 4, this chapter applies the previous chapter's framework to obtain the first optimal survival tree algorithm called *SurTree*. In addition, we show how the depth-two technique by Demirović et al. (2022) can be adapted for survival analysis and we provide a detailed experimental comparison of our new

method with the local search method OST and the greedy heuristic CTree (Hothorn et al., 2006). Our experiments show that SurTree's out-of-sample performance on average is better than CTree and similar to OST while outperforming OST in runtime for realistic cases. Since SurTree is the first optimal method for survival trees, our method also helps assess the quality of heuristic solutions.

The following sections introduce related work and the preliminaries for our work. We then present our DP-based approach to optimal survival trees, evaluate it on synthetic and real datasets, and compare it with the state of the art.

5.2. Related Work

Survival analysis Traditionally, many statistical approaches have been developed for dealing with censored data (Chung et al., 1991). This includes nonparametric approaches, such as the Kaplan-Meier method (Kaplan and Meier, 1958) and the Nelson-Aalen estimator (Nelson, 1972; Aalen, 1978), semiparametric approaches, such as Cox proportional hazards regression (Cox, 1972), and parametric approaches such as linear regression. Wang et al. (2019) provide an overview of the later use of machine learning for survival analysis, including survival trees, random survival forests (Ishwaran et al., 2008), support vector machines (Van Belle et al., 2011), and neural networks (Chi et al., 2007).

Survival trees Many greedy top-down survival tree *splitting criteria* have been proposed. They can be divided into criteria that promote within-node homogeneity or between-node heterogeneity. Examples of the former are Gordon and Olshen (1985), Davis and Anderson (1989), Therneau et al. (1990), and LeBlanc and Crowley (1992). Examples of the latter are Ciampi et al. (1986), Segal (1988), and LeBlanc and Crowley (1993). Other developments are presented by Molinaro et al. (2004), who propose to weigh the uncensored data based on inverse-propensity weighting; Su and Fan (2004), who consider survival analysis for clustered events using a maximum likelihood approach based on frailty models; and Hothorn et al. (2006), who introduce χ^2 tests as stopping criterion to prevent variable selection bias.

Recently, Bertsimas et al. (2022) presented OST (optimal survival trees), based on the coordinate-descent method proposed by Dunn (2018), by iteratively improving one node in the tree until a local optimum is reached. Because the problem is non-convex, they repeat this process several times to increase the probability of finding a good tree. Their results show that local search can outperform greedy heuristics such as those by Therneau et al. (1990) and Hothorn et al. (2006). However, despite its name, OST provides no guarantee of converging to the global optimum.

Since we do not know of any optimal survival tree method, and given the scalability and generalizability of our DP method presented in Chapter 4, this study explores the use of DP for survival trees and the benefit of globally optimizing survival trees.

5.3. Preliminaries

Definitions and notation We aim to optimize a survival tree over a dataset $\mathcal{D}$. This dataset consists of instances that either experienced the event of interest or were

censored. Each of these instances is described by a set of features. The following defines each of these terms:

An *event of interest*, or simply *event* or *death*, is the non-repeatable event for which the time until occurrence is measured within a trial. The *time-to-event* is the amount of time leading up to the event of interest from the beginning of the observation. In this study, we consider *right-censored* data, which means that for some instances the exact time-to-event is unknown, but lower bounded by some known time, for example, because a patient left the trial before the event of interest could be observed.

A dataset $\mathcal{D}$, or a trial, consists of a set of *instances* $(t_i, \delta_i, \mathbf{fv}_i)$, each described by a feature vector $\mathbf{fv}_i$, a *censoring indicator* $\delta_i \in \{0, 1\}$ stating whether the event of interest was observed and a *time* $t_i > 0$. In case of censoring ($\delta_i = 0$), t_i denotes the last time of observation. Otherwise, t_i denotes the time-to-event. The feature vector describes the instance by a set of features $\mathcal{F}$. Our method assumes that all features are binarized beforehand such that each feature is a binary predicate. We write $f \in \mathbf{fv}_i$ if the predicate holds for instance i or $\bar{f} \in \mathbf{fv}_i$ if it does not hold. We use $\mathcal{D}(f_i)$ and $\mathcal{D}(\bar{f}_i)$ to refer to all instances in $\mathcal{D}$ for which the predicate f_i is valid or not, respectively. Multiple feature splits can be stacked so that, for example, $\mathcal{D}(f_1, \bar{f}_2)$ refers to all instances for which f_1 holds and f_2 does not hold.

A *decision tree* partitions instances based on their features. We consider *binary* trees, where each node is either a *decision node* with two children, or a *leaf node*. Each decision node splits the dataset on a certain feature. Each leaf node assigns a label to every instance that ends up at that leaf node. A *survival tree* (see Fig. 5.1) is a special type of decision tree that assigns in each leaf node not just a label, but a survival distribution that describes the survival odds after a certain amount of time.

Survival analysis background The goal of survival analysis is to accurately describe the *survival function*, which gives the probability of survival after a time t, denoted as $S(t) = P(T \geq t)$, with T the true time of the event (Wang et al., 2019). Its opposite is the cumulative death distribution function $F(t) = 1 - S(t)$, with its derivative, the death density function $f(t) = \frac{d}{dt}F(t)$.

One of the most used estimators for the survival function is the Kaplan-Meier estimator (Kaplan and Meier, 1958):

$$\hat{S}(t) = \prod_{t' \leq t} \left(1 - \frac{d(t')}{n(t')}\right), \tag{5.1}$$

with $d(t')$ the number of deaths at time t' and $n(t')$ the number of survivors up and until time t':

$$d(t) = |\{(t_i, \delta_i, \mathbf{fv}_i) \in \mathcal{D} \mid t_i = t \wedge \delta_i = 1\}|, \tag{5.2}$$

$$n(t) = |\{(t_i, \delta_i, \mathbf{fv}_i) \in \mathcal{D} \mid t_i \geq t\}|. \tag{5.3}$$

The *hazard function* (also known by the force of mortality, the instantaneous death rate, or the conditional failure rate), given by $\lambda(t) = \frac{f(t)}{S(t)}$, indicates the frequency (or rate) of the event of interest happening at time t, provided that it has

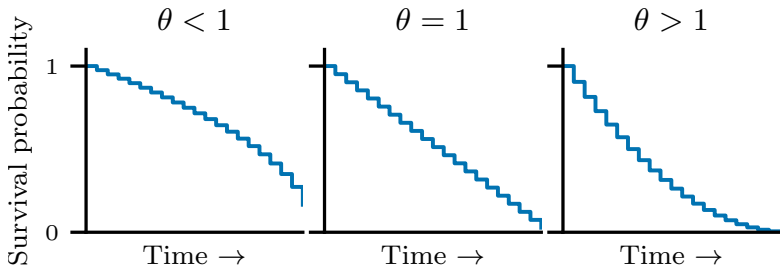


Figure 5.2: A visualization of how θ affects a survival distribution $\hat{S}(t)$. Every plot uses the same $\hat{\Lambda}(t)$, but use $\theta = 0.5$, $\theta = 1$ and $\theta = 2$ respectively.

not happened before time t yet (Dunn and Clark, 2009). Alternatively, it can be written as $\lambda(t) = -\frac{d}{dt} \ln S(t)$. The *cumulative hazard function* is the integral over the hazard function $\Lambda(t) = \int_0^t \lambda(u)du$, and thus the survival function $S(t)$ can be rewritten as

$$S(t) = e^{-\Lambda(t)} . \tag{5.4}$$

A commonly used estimator for the cumulative hazard function is the Nelson-Aalen estimator, which is defined analogously to the Kaplan-Meier estimator (Nelson, 1972; Aalen, 1978):

$$\hat{\Lambda}(t) = \sum_{t' \leq t} \frac{d(t')}{n(t')} . \tag{5.5}$$

The Nelson-Aalen estimator of Eq. (5.5) in combination with Eq. (5.4) is what we will use for our method, as explained in the next section.

5.4. Method

We present *SurTree*, a dynamic programming approach to optimizing survival trees. First, we explain what loss function is minimized. Second, we show how DP can be used to find the global optimum for the loss function. Third, we present a special algorithm for trees of depth two that results in a significant increase in scalability.

5.4.1. Loss Function

The optimization of decision trees requires a target loss function. For computational efficiency, the loss function over a leaf node needs to be independent of other leaf nodes. Therefore, like Bertsimas et al. (2022), we optimize the likelihood method from LeBlanc and Crowley (1992). This method assumes that the survival function S_i for each instance i can be approximated by a *proportional hazard model*, described by multiplying the exponent in Eq. (5.4), that is, the baseline hazard model $\hat{\Lambda}(t)$, as estimated by the Nelson-Aalen estimator in Eq. (5.5), by some parameter θ_i:

$$\hat{S}_i(t) = e^{-\theta_i \hat{\Lambda}(t)} . \tag{5.6}$$

Fig. 5.2 shows how θ_i changes the survival function $S_i(t)$.

LeBlanc and Crowley (1992) show that for a given dataset $\mathcal{D}$ the estimate $\hat{\theta}$ with maximum likelihood is equal to:

$$\hat{\theta} = \frac{\sum_{(t_i,\delta_i,\mathbf{fv}_i)\in\mathcal{D}} \delta_i}{\sum_{(t_i,\delta_i,\mathbf{fv}_i)\in\mathcal{D}} \hat{\Lambda}(t_i)} \,. \tag{5.7}$$

This means that for a single instance i, the saturated coefficient that perfectly maximizes the likelihood for that instance alone is given by

$$\hat{\theta}_i^{sat} = \frac{\delta_i}{\hat{\Lambda}(t_i)} \,. \tag{5.8}$$

The loss for a single instance is then defined as the difference between the log-likelihood of the instance's leaf node $\hat{\theta}$ and the log-likelihood of the instance's $\hat{\theta}_i^{sat}$. In the appendix, we show how this results in the following loss for a dataset $\mathcal{D}$ that ends up in a leaf node with parameter $\hat{\theta}$:

$$\mathcal{L}(\mathcal{D},\hat{\theta}) = \sum_{(t_i,\delta_i,\mathbf{fv}_i)\in\mathcal{D}} \left(\hat{\Lambda}(t_i)\hat{\theta} - \delta_i \log \hat{\Lambda}(t_i) - \delta_i \log \hat{\theta} - \delta_i\right). \tag{5.9}$$

5

5.4.2. Dynamic Programming Approach

The loss function of Eq. (5.9) consists of several nonlinear terms that prevent it from being directly optimized with mixed-integer linear programming. However, it can be optimized with dynamic programming. The key change compared to a DP formulation for standard decision trees is the base case: instead of assigning a class based on the majority vote, we now optimize $\hat{\theta}$ such that the loss is minimized. We apply this to the DP formulation from Demirović et al. (2022):

$$T(\mathcal{D},d,n) = \begin{cases} \min_{\hat{\theta}} \mathcal{L}(\mathcal{D},\hat{\theta}) & n = 0 \\ T(\mathcal{D},d,2^d-1) & n > 2^d-1 \\ T(\mathcal{D},n,n) & d > n \\ \min\{T(\mathcal{D}(\overline{f}),d-1,n-i-1) & \\ \quad +T(\mathcal{D}(f),d-1,i) & \\ \quad : f\in\mathcal{F}, i\in[0,n-1]\} & \text{otherwise.} \end{cases} \tag{5.10}$$

In this equation, subproblems are defined by the dataset $\mathcal{D}$, the (remaining) tree depth d, and branching node budget n. When $n = 0$, a leaf node is returned with a survival distribution given by θ for which the loss is minimized. When the depth or branching node budget exceeds what is possible according to the other budget, for example, when $d > n$, the budgets are updated accordingly. Otherwise, a branching node is optimized by looping over all possible branching features $f \in \mathcal{F}$ and all possible branching node budget distributions. The loss of the two subtrees is summed for each possible split, and the best possible split is returned.

The solutions to the subproblems $\langle\mathcal{D},d,n\rangle$ are *cached*. Cached solutions are also used as lower bounds. Upper bounds (best solution so far) and lower bounds for a subtree search are used to terminate the search early.

To prevent overfitting, we use *hyperparameter tuning* to tune the depth and number of branching nodes. Alternatively, a cost-complexity parameter can be used to penalize adding more branching nodes. However, tuning the depth and number of nodes directly allows to reuse the cache, yielding a speed improvement, without loss of solution quality.

5.4.3. Trees of Depth Two

Demirović et al. (2022) developed a major scalability improvement for optimizing classification trees of maximum depth two. Instead of applying the splitting and recursing technique (as done similarly in Eq. (5.10)), which requires counting class occurrences for every possible leaf node, this algorithm precomputes the class occurrences by looping over all pairs of features in the feature vector $f_i, f_j \in \mathbf{fv}_k$ for each instance k. The counts can then be used to directly compute the misclassification score for each leaf node without having to examine the entire dataset again. Van der Linden et al. (2023) show that this method can also be generalized to other optimization tasks, provided that a per-instance breakdown of the loss can be formulated.

Here, we provide a breakdown of the per-instance contribution to the costs, such that the same precomputing technique can be used for survival analysis. Pseudocode is provided in the appendix.

First, we split Eq. (5.9) into several summations:

$$\hat{\theta} \sum_i \hat{\Lambda}(t_i) - \sum_i \delta_i \log \hat{\Lambda}(t_i) - \log \hat{\theta} \sum_i \delta_i - \sum_i \delta_i \,. \tag{5.11}$$

Then, by substituting Eq. (5.7) into the above formula, we get the following:

$$\begin{aligned} \mathcal{L}(\mathcal{D}, \hat{\theta}) =& \frac{\sum_i \delta_i}{\sum_i \hat{\Lambda}(t_i)} \sum_i \hat{\Lambda}(t_i) - \sum_i \delta_i \log \hat{\Lambda}(t_i) \\ & - \log \left(\frac{\sum_i \delta_i}{\sum_i \hat{\Lambda}(t_i)} \right) \sum_i \delta_i - \sum_i \delta_i \\ =& - \sum_i \delta_i \log \hat{\Lambda}(t_i) - \log \left(\frac{\sum_i \delta_i}{\sum_i \hat{\Lambda}(t_i)} \right) \sum_i \delta_i \,. \end{aligned} \tag{5.12}$$

Eq. (5.12) is expressed as a function of several sums over the instances. These sums can be precomputed in the same way as class occurrences are precomputed in (Demirović et al., 2022). Three sums need to be computed: the *event sum* ES, the

hazard sum HS, and the *negative log hazard sum* NLHS.

$$\mathrm{ES}(f_i, f_j) = \sum_{(t_k, \delta_k, \mathbf{fv}_k) \in \mathcal{D}(f_i, f_j)} \delta_k \tag{5.13}$$

$$\mathrm{HS}(f_i, f_j) = \sum_{(t_k, \delta_k, \mathbf{fv}_k) \in \mathcal{D}(f_i, f_j)} \hat{\Lambda}(t_k) \tag{5.14}$$

$$\mathrm{NLHS}(f_i, f_j) = \sum_{(t_k, \delta_k, \mathbf{fv}_k) \in \mathcal{D}(f_i, f_j)} -\delta_k \log \hat{\Lambda}(t_k) \tag{5.15}$$

Eqs. (5.13)-(5.15) compute the event, hazard, and negative log hazard sum for the leaf node with data that satisfies feature f_i and f_j. The sums for the other leaf nodes can be computed as follows (similarly for HS and NLHS):

$$\mathrm{ES}(\overline{f_i}) = \mathrm{ES} - \mathrm{ES}(f_i)\,, \tag{5.16}$$

$$\mathrm{ES}(f_i, \overline{f_j}) = \mathrm{ES}(f_i) - \mathrm{ES}(f_i, f_j)\,, \tag{5.17}$$

$$\mathrm{ES}(\overline{f_i}, f_j) = \mathrm{ES}(f_j) - \mathrm{ES}(f_i, f_j)\,, \tag{5.18}$$

$$\mathrm{ES}(\overline{f_i}, \overline{f_j}) = \mathrm{ES} - \mathrm{ES}(f_i) - \mathrm{ES}(f_j) + \mathrm{ES}(f_i, f_j)\,. \tag{5.19}$$

Here, ES denotes the event sum over the whole dataset $\mathcal{D}$, while $\mathrm{ES}(f_i)$ denotes the event sum on the dataset $\mathcal{D}(f_i)$. Once these sums are computed for pairs of features, the final loss for each split and each possible leaf node can be computed from the sums:

$$\mathcal{L}(\mathcal{D}) = \mathrm{NLHS} - \mathrm{ES} \log\left(\frac{\mathrm{ES}}{\mathrm{HS}}\right). \tag{5.20}$$

Since we only explicitly count the values for when f_i and f_j hold, and derive the other cases implicitly through Eqs. (5.16)-(5.19), the runtime is reduced from $O(|\mathcal{F}|^2|\mathcal{D}|)$ to $O(m^2|\mathcal{D}|)$, with m the maximum number of features that hold for any instance in $\mathcal{D}$. This is specifically advantageous when features hold sparingly. Non-sparse features are flipped to improve sparsity.

5.5. Experiments

The following introduces the experiment setup, the survival analysis metrics, a scalability analysis with an evaluation of the impact of our depth-two algorithm, and the out-of-sample performance of SurTree and two other methods.

5.5.1. Experiment Setup

Methods We implemented SurTree in C++ with a Python interface using the STreeD framework (Van der Linden et al., 2023).[1] In our experiment setup,[2] we compare SurTree with the Julia implementation of OST (Bertsimas et al., 2022) and the R implementation of CTree (Hothorn et al., 2006). Each method is tuned

[1] `https://github.com/AlgTUDelft/pystreed`
[2] `https://github.com/TimHuisman1703/streed_sa_pipeline`

using ten-fold cross-validation. For SurTree, we tune the depth and node budget. For CTree, we tune the confidence criterion. For OST, we tune the depth and, simultaneously, OST automatically tunes the cost-complexity parameter as part of its training. All experiments were run on an Intel i7-6600U CPU with 4GB RAM with a time-out of 10 minutes.

Data We evaluate both on synthetic data to measure the effect of censoring and of having more data, and on real datasets. The real datasets are taken from the SurvSet repository (Drysdale, 2022). Since the results on the synthetic data show that the differences between the methods are more clearly visible for larger datasets, we limit our real data analysis to datasets with more than 2000 instances. We evaluate out-of-sample performance on the real datasets using five-fold cross-validation.

The synthetic data is generated according to the procedure described by Bertsimas et al. (2022). First, we generate n feature vectors, with three continuous features, one binary feature, and two categorical features with three and five categories. Each of the features is uniformly distributed. Second, we randomly generate a survival tree T of depth five that splits on random features and assign a random distribution to each leaf node (see the appendix for a list of used distributions). Third, for each of the n instances, we classify the instance using the tree and assign it a random time-to-event t_i by sampling from the corresponding leaf distribution. After that, we assign the instance a random value u_i, uniformly distributed between 0 and 1. Fourth, we choose the lowest value for k such that for at most $c \cdot 100\%$ of the observations, $k(1-u_i^2) < t_i$ holds. Finally, for each observation for which $k(1-u_i^2) < t_i$, we set $t_i = k(1-u_i^2)$ and $\delta_i = 0$. For every other observation, we leave t_i and set $\delta_i = 1$.

We evaluate each method with a depth limit of four on five generated datasets for each combination of $n \in \{100, 200, 500, 1000, 2000, 5000\}$ and $c \in \{0.1, 0.5, 0.8\}$, each with a corresponding test set of 50,000 instances.

Preprocessing We use one-hot encoding to encode categorical variables. For categorical variables with more than ten categories, the least frequent categories or combined into an 'other' category. Because the dynamic programming approach of SurTree requires binary features, we binarize the numeric features using ten quantiles on all possible thresholds. Identical features and binary features that identify less than 1% of the data are removed. We evaluate CTree and OST on the numeric data and SurTree on the binarized data.

5.5.2. Survival Metrics

To evaluate the out-of-sample performance of all methods, we compare each method using two common metrics from the literature: *Harrell's C-index* (H_C) (Harrell et al., 1982), and the integrated Brier score (IB) (Graf et al., 1999).

Harrell's C-index Harrell's C-index measures the concordance score. Two instances are (dis)concordant if an earlier known death ($t_i < t_j$) for one instance means a (lower) higher risk of death for that instance ($\theta_i > \theta_j$). When $\theta_i = \theta_j$, the pair is said to have a *tied risk*. Since for censored observations, the time-to-event is not known, we can only compare pairs for which the instance with an *earlier time* is

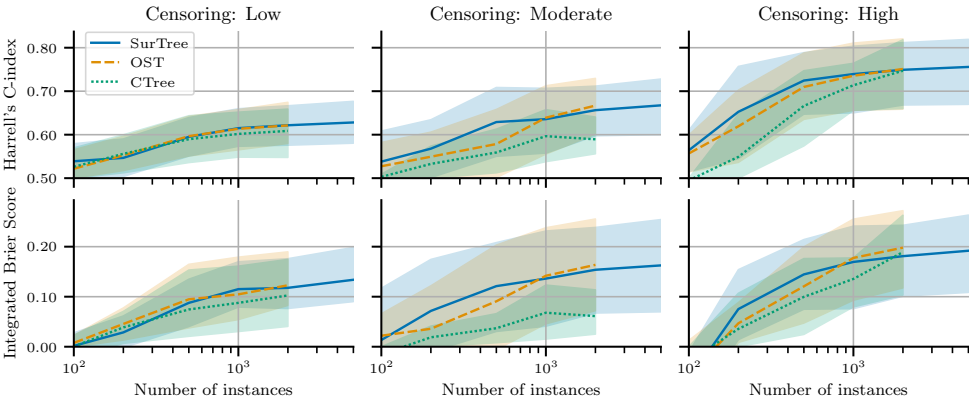


Figure 5.3: Harrell's C-index and the integrated Brier score on the synthetic datasets (except time-outs).

5

not censored. The number of concordant, discordant, and tied-risk pairs can be calculated using the following formulas respectively:

$$CC = \sum_{i,j} \mathbb{1}(t_i < t_j)\mathbb{1}(\theta_i > \theta_j)\delta_i \,, \tag{5.21}$$

$$DC = \sum_{i,j} \mathbb{1}(t_i < t_j)\mathbb{1}(\theta_i < \theta_j)\delta_i \,, \tag{5.22}$$

$$TR = \sum_{i,j} \mathbb{1}(t_i < t_j)\mathbb{1}(\theta_i = \theta_j)\delta_i \,. \tag{5.23}$$

Harrell's C-index is computed as follows:

$$H_C = \frac{CC + 0.5 \cdot TR}{CC + TR + DC} \,. \tag{5.24}$$

The advantage of Harrell's C-index is that it does not make any parametric assumptions and that it works well for the proportional hazard model as used in this chapter. Its disadvantage is that it does not take incomparable pairs into account, which is a problem when censoring is high. Note that a random predictor has an expected score of $H_C = 0.5$.

Integrated Brier score The Brier score (Brier, 1950) is commonly used to evaluate probability forecasts, and measures the mean square prediction error. This measure can be used to evaluate survival distributions at a specific point in time. For evaluating the whole distribution, Graf et al. (1999) developed the integrated Brier score:

$$IB = \frac{\sum_i \int_{\underline{t}}^{t_i} \frac{(1-\hat{S}_i(t))^2}{\hat{G}(t)} dt + \delta_i \int_{t_i}^{\bar{t}} \frac{(\hat{S}_i(t))^2}{\hat{G}(t_i)} dt}{|\mathcal{D}|(\bar{t} - \underline{t})} \,. \tag{5.25}$$

The integrated Brier score evaluates the Brier score over a time interval, with each time step weighed by the Kaplan-Meier estimator of the censoring distribution $\hat{G}(t)$. We compute the integrated Brier score using the test data over the time periods that fall within the 10% and 90% quantile of t_i in the test data, given by $\underline{t}$

Dataset	$\lvert\mathcal{D}\rvert$	Censoring (%)	$\lvert\mathcal{F}_{num}\rvert$	$\lvert\mathcal{F}\rvert$	Harrell's C-index			Integrated Brier Score		
					CTree	OST	SurTree	CTree	OST	SurTree
Aids2	2839	38.0%	4	22	**0.53**	**0.53**	**0.53**	**0.01**	**0.01**	0.00
Dialysis	6805	76.4%	4	35	0.64	0.65	**0.66**	0.07	**0.09**	0.08
Framingham	4658	68.5%	7	60	0.67	0.67	**0.68**	0.09	**0.10**	**0.10**
Unempdur	3241	38.7%	6	45	**0.70**	0.69	0.69	**0.11**	0.10	0.10
Acath	2258	34.0%	3	21	0.59	0.58	**0.60**	**0.03**	0.02	**0.03**
Csl	2481	89.1%	6	42	**0.78**	0.76	0.75	**0.10**	**0.10**	**0.10**
Datadivat1	5943	83.6%	5	21	0.63	**0.64**	0.63	**0.08**	0.05	0.06
Datadivat3	4267	94.4%	7	30	0.65	0.63	**0.66**	0.02	0.02	**0.03**
Divorce	3371	69.4%	3	5	0.52	**0.53**	**0.53**	0.01	**0.02**	**0.02**
Flchain	6524	69.9%	10	60	**0.92**	**0.92**	**0.92**	0.65	0.65	**0.66**
Hdfail	52422	94.5%	6	27	-	-	**0.81**	-	-	**0.41**
Nwtco	4028	85.8%	7	17	**0.70**	**0.70**	0.69	0.12	**0.13**	**0.13**
Oldmort	6495	69.7%	7	33	0.64	**0.65**	0.63	**0.06**	0.05	0.05
Prostatesurv.	14294	94.4%	3	8	**0.75**	**0.75**	**0.75**	0.09	**0.10**	**0.10**
Rott2	2982	57.3%	11	50	0.68	0.68	**0.69**	0.12	**0.15**	0.14
Wins per metric					6	7	**10**	6	8	**9**
Average rank					2.07	2.03	**1.83**	2.21	1.97	**1.77**

Table 5.1: Out-of-sample Harrell's C-index and integrated Brier score for datasets from SurvSet (Drysdale, 2022) for trees of maximum depth $d = 3$. $|\mathcal{F}_{num}|$ is the number of original features. $|\mathcal{F}|$ is the resulting number of binarized features.

and $\bar{t}$ respectively. For easier comparison, we report the normalized relative score $(1 - IB/IB_0)$, with IB_0 the score obtained from the Kaplan-Meier estimator over the whole dataset.

The integrated Brier score also makes no parametric assumptions on the data. Another advantage is that it considers both the censored and non-censored data.

5.5.3. Scalability

Synthetic data To evaluate the scalability of each method, we compare the runtime of each algorithm for increasing maximum depth and features. Each method is evaluated twice for five synthetic datasets with $n = 5000$ and $c = 0.5$. Once for the original setting ($f = 1$) with three continuous, one binary, and two categorical features, and once with double the number of features ($f = 2$): six continuous, two binary, and four categorical features. After binarization, this results in 39 and 78 binary features, respectively.

Fig. 5.4 shows that for $f = 1$ up to depth 5, and $f = 2$ up to depth 4, SurTree has a lower runtime than OST. SurTree's runtime scales exponentially with increasing maximum depth, whereas OST's runtime scales linearly. OST has a relatively high runtime for low depth because it randomly restarts its local search several times (by default 100 times) to improve the quality of the solution. In contrast, SurTree immediately finds the globally optimal solution. CTree surprisingly has approximately a constant runtime for increasing depth and number of features.

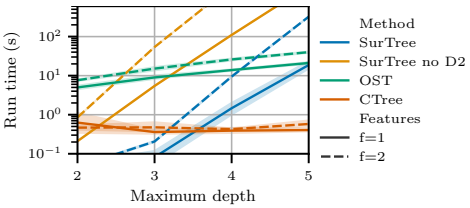


Figure 5.4: Runtime performance for increasing depth, for 3 continuous, 1 binary, and 2 categorical features ($f = 1$) or 6 continuous, 2 binary, and 4 categorical features ($f = 2$).

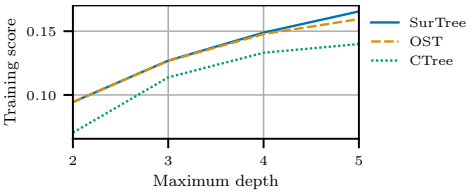


Figure 5.5: Normalized training loss for CTree, OST, and SurTree, when trained with binarized data.

Real data Despite SurTree being an optimal method, SurTree's average runtime for optimizing trees of maximum depth three for the real datasets (including hyper-tuning) is lower than both CTree and OST. On average, it is more than 100 times faster than OST (geometric mean performance ratio). CTree's worse performance here must be attributed to the cross-validation method.

Depth-two algorithm Fig. 5.4 also shows the increase in scalability due to our algorithm for trees of depth two. On average, the depth-two algorithm reduces runtime 45 times (geometric mean, not considering time-outs).

5.5.4. Out-of-Sample Results

Synthetic data Fig. 5.3 shows the performance on the synthetic data for an increasing number of instances. The results are split for low, moderate, and high censoring. Since Harrell's C-index only measures performance for instances that are comparable, Harrell's C-index is slightly higher for high censoring. In general, each method performs better with more data, but both OST and CTree time out when $n = 5000$. Furthermore, these results show that both OST and SurTree perform significantly better than CTree, specifically for moderate and high censoring. OST and SurTree perform similarly, but a Wilcoxon signed rank test shows that SurTree has a better Harrell's C-index than OST for moderate and high censoring and few instances ($n = 200, 500$).

Real data Table 5.1 shows the out-of-sample H_C and IB scores for trees with a maximum depth of three. A Wilcoxon signed rank test reveals that both OST and SurTree perform significantly better (95% confidence) than CTree on both Harrell's C-index and the integrated Brier score. On average, SurTree performs slightly better than OST, but this difference is not statistically significant. Both OST and CTree resulted in one time-out (for Hdfail). For CTree, this is the result of a slow cross-validation algorithm in R based on Harrell's C-index that requires a quadratic number of comparisons. The appendix also shows results for trees with a maximum depth of four.

Training score Since SurTree is the first optimal survival tree method, we can now measure in reasonable time how far non-optimal methods are from the optimal

solution, when comparing training scores on the same (binarized) data. Fig. 5.5 compares the mean training score of CTree, OST, and SurTree on five synthetic training datasets with $n = 5000$ and $c = 0.5$, without hyper-tuning. In this figure, the training score is the normalized loss, with 0 referring to the loss of a single leaf node and 1 referring to zero loss. The difference between SurTree and OST is largest for $d = 5$, where SurTree's training score is 4% better than OST's. The difference between SurTree and CTree is largest for $d = 2$, where SurTree's training score is 34% higher than CTree's.

5.6. Conclusion

We present SurTree, the first survival tree method with global optimality guarantees. The out-of-sample comparison shows it performs better than an existing greedy heuristic and similar to a state-of-the-art local search approach. SurTree uses dynamic programming and a special algorithm for trees of depth two resulting in runtimes even lower than the state-of-the-art local search method that does not provide optimality guarantees.

To improve the prediction quality, future work could explore the effect of fitting a Cox proportional hazards model in each leaf node, instead of only a single constant proportional hazard parameter (Cox, 1972).

Appendices for Chapter 5

5.A. Loss Function

Let $h \in T$ be the leaf nodes of a tree T, $\hat{\theta}_h$ the θ estimate for leaf node h and $\mathcal{D}_h$ the instances that end up in this leaf node. Then, as done by LeBlanc and Crowley (1992), the likelihood function of a survival tree can be formulated as

$$L = \prod_{h \in T} \prod_{(t_i, \delta_i, \mathbf{fv}_i) \in \mathcal{D}_h} (\hat{\theta}_h \hat{\lambda}(t_i))^{\delta_i} e^{-\hat{\theta}_h \hat{\Lambda}(t_i)} . \tag{5.26}$$

Here $\hat{\Lambda}(t)$ is an estimate of the baseline cumulative hazard function. As similarly done by Bertsimas et al. (2022), we estimate it using the Nelson-Aalen estimator, which is a first iteration estimator according to LeBlanc and Crowley (1992).

5

Eq. (5.26) results in the partial log-likelihood function LL for a single leaf node:

$$LL(\mathcal{D}, \hat{\theta}) = \sum_{(t_i, \delta_i, \mathbf{fv}_i) \in \mathcal{D}} \delta_i \log \hat{\lambda}(t_i) + \delta_i \log \hat{\theta} - \hat{\Lambda}(t_i)\hat{\theta} . \tag{5.27}$$

Taking the derivative with respect to $\hat{\theta}$ results in the following:

$$\frac{dLL(\mathcal{D}, \hat{\theta})}{d\hat{\theta}} = \sum_{(t_i, \delta_i, \mathbf{fv}_i) \in \mathcal{D}} \left(\frac{\delta_i}{\hat{\theta}} - \hat{\Lambda}(t_i) \right) . \tag{5.28}$$

Setting the derivative to zero yields the maximum likelihood value for $\hat{\theta}$ in this leaf node:

$$\hat{\theta} = \frac{\sum_{(t_i, \delta_i, \mathbf{fv}_i) \in \mathcal{D}} \delta_i}{\sum_{(t_i, \delta_i, \mathbf{fv}_i) \in \mathcal{D}} \hat{\Lambda}(t_i)} . \tag{5.29}$$

When the dataset consists of only one instance i, Eq. (5.29) gives the *saturated* coefficient $\hat{\theta}_i^{sat}$ that perfectly maximizes the likelihood for this instance alone:

$$\hat{\theta}_i^{sat} = \frac{\delta_i}{\hat{\Lambda}(t_i)} . \tag{5.30}$$

The log-likelihood for one instance is given by Eq. (5.27) when the dataset consists of one instance. By this equation, the loss for a single instance i is then defined as the difference between the log-likelihood of the instance's $\hat{\theta}_i^{sat}$ and the log-likelihood of the instance's leaf node $\hat{\theta}$.

$$\begin{aligned}\mathcal{L}(\hat{\theta}) =& \left(\delta_i \log \hat{\lambda}(t_i) + \delta_i \log \frac{\delta_i}{\hat{\Lambda}(t_i)} - \hat{\Lambda}(t_i)\frac{\delta_i}{\hat{\Lambda}(t_i)}\right) \\ & - \left(\delta_i \log \hat{\lambda}(t_i) + \delta_i \log \hat{\theta} - \hat{\Lambda}(t_i)\hat{\theta}\right) \\ =& \hat{\Lambda}(t_i)\hat{\theta} + \delta_i \log \frac{\delta_i}{\hat{\Lambda}(t_i)} - \delta_i - \delta_i \log \hat{\theta} \\ =& \hat{\Lambda}(t_i)\hat{\theta} - \delta_i \log \hat{\Lambda}(t_i) - \delta_i \log \hat{\theta} - \delta_i \,.\end{aligned} \tag{5.31}$$

The last step in this derivation uses the fact that δ_i is binary, and therefore $\delta_i \log \delta_i$ is always zero (under the common simplification that $0 \log 0$ is zero). Summing the loss of the instances in a leaf node yields the loss function in the main text:

$$\mathcal{L}(\mathcal{D}, \hat{\theta}) = \sum_{(t_i, \delta_i, \mathbf{fv}_i) \in \mathcal{D}} \left(\hat{\Lambda}(t_i)\hat{\theta} - \delta_i \log \hat{\Lambda}(t_i) - \delta_i \log \hat{\theta} - \delta_i\right). \tag{5.32}$$

To prevent infinite loss when $\hat{\theta} = 0$, LeBlanc and Crowley (1992) set

$$\hat{\theta} = 1/(2 \sum_i \hat{\Lambda}(t_i)) \tag{5.33}$$

when no events are recorded in a leaf node. The loss for the whole tree can be computed by summing the loss of each leaf node.

5.B. Depth-Two Solver

Let X denote the cost tuple $(\mathrm{ES}, \mathrm{HS}, \mathrm{NLHS})$ for the depth-two algorithm, consisting of the *event sum* ES, the *hazard sum* HS and the *negative log hazard sum* NLHS. Let $C(\mathrm{ES}, \mathrm{HS}, \mathrm{NLHS})$ compute the loss from the tuple X:

$$C(\mathrm{ES}, \mathrm{HS}, \mathrm{NLHS}) = \mathrm{NLHS} - \mathrm{ES} \log \left(\frac{\mathrm{ES}}{\mathrm{HS}}\right). \tag{5.34}$$

Then the depth-two algorithm is given by the pseudo-code in Algorithm 8. In this algorithm, the X tuples are summed using element-wise addition. The first step is the pre-computation of the X tuples. The second step is going over all possible branching nodes and computing the loss from the pre-computed values. The final step is to return the loss of the root node that has the minimal sum of left and right loss.

The values for X that are not pre-computed can be derived using the following formulas:

$$X(\overline{f_i}) = X - X(f_i)\,, \tag{5.35}$$

$$X(f_i, \overline{f_j}) = X(f_i) - X(f_i, f_j)\,, \tag{5.36}$$

$$X(\overline{f_i}, f_j) = X(f_j) - X(f_i, f_j)\,, \tag{5.37}$$

$$X(\overline{f_i}, \overline{f_j}) = X - X(f_i) - X(f_j) + X(f_i, f_j)\,. \tag{5.38}$$

Algorithm 8: First, pre-compute the tuples X describing the event, hazard, and negative log hazard sum. Then find for every feature the best loss for the left and right subtree $BestL.\mathcal{L}$ and $BestR.\mathcal{L}$. Finally, return the loss of the best tree of at most depth two.

$X \leftarrow (0, 0, 0)$
$X(f_i) \leftarrow (0, 0, 0) \quad \forall f_i \in \mathcal{F}$
$X(f_i, f_j) \leftarrow (0, 0, 0) \quad \forall f_i, f_j, \in \mathcal{F}$ s.t. $i < j$
for $(t, \delta, \mathbf{fv}) \in \mathcal{D}$ **do**
  $X \leftarrow X + (\delta, \hat{\Lambda}(t), -\delta \log \hat{\Lambda}(t))$
  for $f_i \in \mathbf{fv}$ **do**
    $X(f_i) \leftarrow X(f_i) + (\delta, \hat{\Lambda}(t), -\delta \log \hat{\Lambda}(t))$
    for $f_j \in \mathbf{fv}$, *s.t.* $i < j$ **do**
      $X(f_i, f_j) \leftarrow X(f_i, f_j) + (\delta, \hat{\Lambda}(t), -\delta \log \hat{\Lambda}(t))$
for $f_i \in \mathcal{F}$ **do**
  for $f_j \in \mathcal{F}$ **do**
    $\mathcal{L}_L = C(X(\bar{f}_i, f_j)) + C(X(\bar{f}_i, \bar{f}_j))$
    $\mathcal{L}_R = C(X(f_i, f_j)) + C(X(f_i, \bar{f}_j))$
    if $BestL.\mathcal{L}(f_i) > \mathcal{L}_L$ **then**
      $BestL.\mathcal{L}(f_i) \leftarrow \mathcal{L}_L$
    if $BestR.\mathcal{L}(f_i) > \mathcal{L}_R$ **then**
      $BestR.\mathcal{L}(f_i) \leftarrow \mathcal{L}_R$
return $\min_{f_i \in \mathcal{F}} BestL.\mathcal{L}(f_i) + BestR.\mathcal{L}(f_i)$

5.C. Distributions in Ground Truth Trees

To determine the time-to-event for each instance during the synthetic dataset generation, a ground truth tree is generated for each dataset. As also done by Bertsimas et al. (2022), each leaf node is assigned a random distribution from the list below. Each option has an equal probability of being assigned.

- Exponential(λ), with $\lambda \in \{0.3, 0.4, 0.6, 0.8, 0.9, 1.15, 1.5, 1.8\}$
- Weibull(k, λ), with $(k, \lambda) \in \{(0.8, 0.4), (0.9, 0.5), (0.9, 0.7), (0.9, 1.1), (0.9, 1.5), (1.0, 1.1), (1.0, 1.9), (1.3, 0.5)\}$
- Lognormal(μ, σ^2), with $(\mu, \sigma^2) \in \{(0.1, 1), (0.2, 0.75), (0.3, 0.3), (0.3, 0.5), (0.3, 0.8), (0.4, 0.32), (0.5, 0.3), (0.5, 0.7)\}$
- Gamma(k, θ), with $(k, \theta) \in \{(0.2, 0.75), (0.3, 1.3), (0.3, 2.0), (0.5, 1.5), (0.8, 1.0), (0.9, 1.3), (1.3, 0.9), (1.5, 0.7)\}$

5.D. Extended Experiment Results

Table 5.2 shows the out-of-sample performance of each method for trees up to depth four. A Wilcoxon signed rank test points out that no significant differences between

	Harrell's C-index			Integrated Brier Score		
Dataset	CTree	OST	SurTree	CTree	OST	SurTree
Aids2	**0.53**	**0.53**	**0.53**	**0.01**	**0.01**	0.00
Dialysis	0.64	**0.67**	0.65	0.07	**0.12**	0.08
Framingham	**0.68**	0.67	0.67	**0.11**	0.09	0.10
Unempdur	0.69	0.69	**0.70**	**0.11**	**0.11**	**0.11**
Acath	**0.59**	**0.59**	**0.59**	0.02	0.02	**0.03**
Csl	**0.78**	0.76	0.76	0.10	0.10	**0.11**
Datadivat1	0.63	**0.64**	0.63	**0.08**	0.05	**0.08**
Datadivat3	**0.66**	0.64	0.65	0.02	**0.03**	**0.03**
Divorce	0.52	**0.53**	**0.53**	0.01	**0.02**	**0.02**
Flchain	**0.92**	**0.92**	**0.92**	**0.66**	**0.66**	**0.66**
Hdfail	-	-	**0.84**	-	-	**0.44**
Nwtco	0.70	**0.71**	0.70	**0.14**	**0.14**	0.13
Oldmort	**0.64**	-	**0.64**	**0.06**	-	**0.06**
Prostatesurvival	**0.76**	**0.76**	**0.76**	0.10	**0.11**	**0.11**
Rott2	0.68	0.68	**0.69**	0.12	**0.15**	**0.15**
Wins per metric	8	8	**9**	7	9	**11**
Average rank	2.07	2.07	**1.87**	2.23	2.03	**1.73**

Table 5.2: Out-of-sample Harrell's C-index and integrated brier score for datasets from SurvSet (Drysdale, 2022) for trees of maximum depth $d = 4$.

the methods can be observed, except for one: SurTree has a significantly better integrated Brier score than CTree ($p < 5\%$).

6

Piecewise Constant and Linear Regression

Preface This chapter applies the general framework of Chapter 4 to regression trees with three different kinds of leaf node functions: constant predictors, simple linear regression, and (normal) linear regression. Trees with (simple) linear regression leaf nodes generalize linear regression and add a nonlinear component. As in Chapter 5, we show how the loss function for the first two cases can be broken down so that the depth-two subroutine presented in Chapter 4 can be used to improve scalability.

Abstract Regression trees are a human-comprehensible machine-learning model that can represent complex relationships. They are typically trained using greedy heuristics because computing optimal regression trees is NP-hard. Contrary to this standard practice, we consider optimal methods and improve the scalability of optimal methods by developing three new dynamic programming approaches. First, we improve the performance of a piecewise constant regression tree method using a special algorithm for trees of depth two. Second, we provide the first optimal dynamic programming method for piecewise multiple linear regression. Third, we develop the first optimal method for piecewise simple linear regression, for which we also provide a special algorithm for trees of depth two. The experimental results show that our methods improve scalability by one or more orders of magnitude over the state-of-the-art optimal methods while performing similarly or better in out-of-sample performance.

6.1. Introduction

Regression trees generalize linear regression by splitting the data before learning a regression model, as seen in Fig. 6.1. This makes regression trees a powerful tool for

Parts of this chapter have been published in Van den Bos, Van der Linden, and Demirović (2024), "Piecewise Constant and Linear Regression Trees: An Optimal Dynamic Programming Approach", in *Proceedings of ICML-24*. This paper resulted from Mim van den Bos' bachelor thesis project, supervised by Emir Demirović and me. Mim van den Bos was responsible for most of the implementation and investigation. I did most of the conceptualization, methodology, and writing.

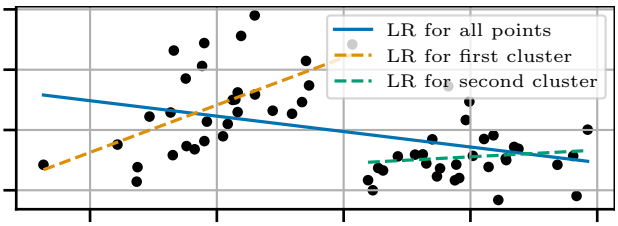


Figure 6.1: The blue line represents standard linear regression (LR), whereas orange and green show the advantage of adding one split before linear regression.

regression analysis, with wide applications ranging from ecology analysis (De'ath and Fabricius, 2000) to clinical psychology (King and Resick, 2014). Because of their rule-based nature, they satisfy the need for complex, nonlinear models that are human-comprehensible, specifically if we learn small, optimal models (Freitas, 2014; Loh, 2014; Rudin, 2019; Carrizosa et al., 2021).

However, most optimal decision tree methods consider only classification. Among the exceptions are Dunn (2018) and Verwer and Zhang (2017) who show how their MIP models can also be used for regression. Dunn (2018) considers both piecewise constant regression trees (with a constant predictor in each leaf node) and piecewise linear regression trees (with a linear predictor in each leaf node, as seen in Fig. 6.1), whereas Verwer and Zhang (2017) only consider constant predictors. However, Dunn (2018) observes that lack of scalability prevents the use of MIP for practical applications. Zhang et al. (2023) are the first to apply DP, but they only consider piecewise constant regression trees. Moreover, they do not incorporate some of the latest scalability improvements for optimal classification trees.

Therefore, this chapter proposes three novel contributions, building on the framework presented in Chapter 4. First, for piecewise constant regression trees, we adapt the depth-two technique by Demirović et al. (2022) to regression to improve scalability over an existing optimal DP regression tree approach. Second, we provide the first optimal DP algorithm for piecewise linear regression trees. Third, we consider the special case of piecewise *simple* linear regression, which restricts each linear model to only one independent variable, and provide another depth-two algorithm that also yields large scalability improvements.

As before, we train *binary* decision trees and assume all possible tree predicates to be given, i.e., in the form of binary features. The linear models in the leaf nodes are trained with the original continuous features.

We compare our methods with ten methods from the literature, both heuristics and optimal methods. Our methods surpass previous optimal methods by one or more orders of magnitude in scalability, mainly because of our new depth-two algorithms. Our methods' out-of-sample performance is on par with or better than the state of the art.

6.2. Related Work

Regression tree heuristics Since computation of optimal decision trees is NP-hard (Hyafil and Rivest, 1976), heuristics are commonly used for computing regression trees. The first regression tree heuristics was AID (Morgan and Sonquist, 1963) which greedily partitions a node such that the sum of squared errors (SSE) is minimized. AID halts when no partition exists which results in an improvement above a certain threshold. CART (Breiman et al., 1984) is one of the most-used heuristics for piecewise constant regression trees. Instead of respecting a minimum improvement threshold, CART grows trees that overfit on the data and then uses cross-validation to prune overfitting branches. GUIDE (Loh, 2002) is a more recent greedy heuristic for both piecewise constant and piecewise (simple) linear trees that uses χ^2-tests to address the bias of preferring to branch on features with many unique values. Other examples of commonly used greedy heuristics for piecewise linear and polynomial regression trees are M5 (Quinlan, 1992; Wang and Witten, 1997), and MARS (Friedman, 1991), respectively.

Apart from greedy heuristics, Hemmateenejad et al. (2011) and Grubinger et al. (2014) propose to use ant colony optimization and evolutionary algorithms. Yang et al. (2016) greedily choose a single continuous splitting feature for the whole tree, and then use MIP on this tree to globally optimize a piecewise linear model for each leaf node. Bertsimas et al. (2021) use Adam (Kingma and Ba, 2015) for gradient optimization of tree splits and then apply polynomial ridge regression in the leaf nodes. Blanquero et al. (2022) use randomized splits to transform the problem into a nonlinear continuous optimization problem with better scalability, but with reduced interpretability. Dunn (2018) uses local search with random restarts to iteratively optimize a single node in the tree. Similarly, Yang et al. (2017) use local search to iteratively optimize a single split decision together with the leaf node regression function using MIP.

Optimal regression trees Early optimal approaches focused on *classification* (see Chapter 2). Optimal MIP models for *regression* trees were proposed by Bertsimas et al. (2017) and Verwer and Zhang (2017). Dunn (2018) presents an optimal MIP method for both piecewise constant and linear trees, but observes the lack of scalability and then proposes to use a local search heuristic instead. The only optimal DP-based approach for piecewise constant regression trees is OSRT (Zhang et al., 2023). They observe that for each subtree with at most k nodes, the error of an optimal k-means clustering provides a lower bound for the current subtree, which can be computed efficiently by using the algorithm by Wang and Song (2011) and Song and Zhong (2020).

Summary Only a few of the many regression tree methods are optimal and among those, scalability remains challenging. Furthermore, no scalable optimal piecewise linear regression method exists yet. We address these gaps by improving the scalability of piecewise constant regression methods and by providing the first optimal DP approaches for piecewise simple and multiple linear regression trees.

6.3. Preliminaries

This section introduces the notation, provides a formal problem definition, and explains the basics of the DP approach.

Notation Let $\mathcal{F}$ be a set of features, $\mathcal{F}_b$ a corresponding set of binarized features, and $\mathcal{D}$ a dataset of instances (x, b, y). The vector $x \in \mathbb{R}^{|\mathcal{F}|}$ represents the continuous feature vector and $b \subseteq \mathcal{F}_b$ represents the binary feature vector, where $f \in b$ means the binary predicate f is satisfied in vector b. Similarly, let $\bar{f}$ represent $f \notin b$. The symbol $y \in \mathbb{R}$ represents the label of an instance. Datasets can be split on binary features f into two datasets $\mathcal{D}_f$ and $\mathcal{D}_{\bar{f}}$ that respectively contain all instances (x, b, y) that satisfy $(f \in b)$ or do not satisfy $(f \notin b)$ feature f. Similarly, we define $\mathcal{D}_{f_i, f_j}$ as the subset of $\mathcal{D}$ that satisfies both f_i and f_j. Let $\bar{y}$ denote the mean of the labels of a dataset $\mathcal{D}$.

Problem definition A binary regression tree is a function $\tau : \mathbb{R}^{|\mathcal{F}|} \times \mathcal{P}(\mathcal{F}_b) \to \mathbb{R}$ that maps an instance (x, b) to a predicted output label $\hat{y}$. Here $\mathcal{P}$ denotes the power set. Each branching node performs a binary feature test from $\mathcal{F}_b$ and directs instances to the left or right subtree. Leaf nodes assign output labels to every instance that ends up in it. The quality of a decision tree is determined by computing the error, such as the Sum of Squared Errors (SSE), of the tree output with respect to the true values.

Optimal regression trees provably globally optimize the SSE for a given size limit (number of nodes or depth of the tree). To prevent overfitting, we penalize the size of the tree, given by the number of branching nodes $N(\tau)$, by a regularization parameter λ, scaled by the total sum of squares of the training data. Therefore, given a maximum depth d and a training dataset $\mathcal{D}$, the objective function is

$$\min_{\tau} \sum_{(x,b,y)\in\mathcal{D}} (y - \tau(x, b))^2 + \lambda N(\tau)\,. \tag{6.1}$$

Dynamic programming approach The basis of the dynamic programming approach is the recursive formulation of the problem. A simplified version of the recursive formulation for minimizing misclassification score by Demirović et al. (2022) is as follows:

$$T(\mathcal{D}, d) = \begin{cases} \min_{\hat{k}\in\mathcal{K}} \sum_{(b,k)\in\mathcal{D}} \mathbb{1}(k \neq \hat{k}) & \text{if } d = 0 \\ \min_{f\in\mathcal{F}_b} \{T(\mathcal{D}_f, d-1) + \\ \qquad T(\mathcal{D}_{\bar{f}}, d-1)\} & \text{if } d > 0 \end{cases} \tag{6.2}$$

In this equation, the maximum depth of the tree d and the dataset $\mathcal{D}$ define the DP state. $\mathcal{K}$ is the set of labels, k the true label and $\hat{k}$ the predicted label. At each branching node $(d > 0)$, all possible branching decisions $f \in \mathcal{F}_b$ are considered, resulting in two independent subproblems each. Leaf nodes $(d = 0)$ select the label with the lowest misclassification costs. For brevity of notation, this and subsequent DP formulations only consider complete trees.

Demirović et al. (2022) significantly enhance scalability by introducing a specialized algorithm for depth-two trees. Instead of using the default recursion, this

algorithm efficiently precomputes class occurrences for the whole dataset and all possible depth-two splits, to prevent traversing the whole dataset repeatedly for computing the misclassification score for each possible leaf node. In the next section, we describe how a similar idea can be used to significantly improve computation of regression trees up to depth two.

6.4. Piecewise Constant Regression Trees

This section explains how optimal piecewise constant regression trees can be computed using DP and how a special solver for trees of depth two greatly improves scalability.

Dynamic programming formulation The following equation adapts Eq. (6.2) for regression:

$$T(\mathcal{D}, d) = \begin{cases} \sum_{(b,y)\in\mathcal{D}} (y - \bar{y})^2 & \text{if } d = 0 \\ \min\limits_{f\in\mathcal{F}_b} \{T(\mathcal{D}_f, d-1) + \\ \qquad T(\mathcal{D}_{\bar{f}}, d-1) + \lambda\} & \text{if } d > 0 \end{cases} \tag{6.3}$$

The updated formulation selects the mean $\bar{y}$ as the leaf node's label and returns the SSE as its cost. Adding a branching node is penalized with a regularization parameter λ. See Appendix 6.A for the full pseudocode including non-complete trees, caching, and bound-based pruning.

Precomputing per-instance costs As introduced in the preliminaries, Demirović et al. (2022) obtain a major increase in scalability because of their special solver for trees of maximum depth two. This is obtained by precomputing class occurrences before looping over every possible combination of feature splits. As a result, instead of traversing the whole dataset for every combination of two features, the precomputation only considers the features that are present for every instance. This algorithm is shown to be very effective for classification, and Van der Linden et al. (2023) generalize this special depth-two solver to any optimization task, provided that the costs for a leaf node can be expressed as a function of the per-instance contribution to the costs.

However, for regression, the error of each instance depends on the mean of all instances in a leaf node, and when one instance is added to or removed from a leaf node, the error of every other instance changes. Despite this, we can rewrite the SSE as a function of three sums over the instances: the sum of y, the sum of y^2, and the number of instances $|\mathcal{D}|$:

$$\sum_{(b,y)\in\mathcal{D}} (y - \bar{y})^2 = \sum_{(b,y)\in\mathcal{D}} y^2 - \frac{\left(\sum_{(b,y)\in\mathcal{D}} y\right)^2}{|\mathcal{D}|}\,. \tag{6.4}$$

For example, with labels 4, 5, and 6, we have $\bar{y} = 5$, and the SSE is 2. Eq. (6.4) yields $4^2+5^2+6^2 = 77$, $(4+5+6)^2 = 225$, and $|\mathcal{D}| = 3$, so $77 - 225/3 = 2$. Therefore, Eq. (6.4) allows us to define the per-instance costs as a three-tuple $(y, y^2, 1)$, which can be summed for multiple instances using element-wise addition. Let P represent

Algorithm 9: Depth-two tree search for a dataset $\mathcal{D}$ and a feature set $\mathcal{F}_b$. $BestL.C$ and $BestR.C$ are the best left and right subtrees respectively.

$Q(f_i) \leftarrow (0,0,0) \quad \forall f_i \in \mathcal{F}_b$
$Q(f_i, f_j) \leftarrow (0,0,0) \quad \forall f_i, f_j, \in \mathcal{F}_b$ s.t. $i < j$
for $(b, y) \in \mathcal{D}$ **do**
  for $f_i \in b$ **do**
    $Q(f_i) \leftarrow Q(f_i) + (y, y^2, 1)$
    for $f_j \in b$, *s.t.* $i < j$ **do**
      $Q(f_i, f_j) \leftarrow Q(f_i, f_j) + (y, y^2, 1)$
for $f_i \in \mathcal{F}_b$ **do**
  for $f_j \in \mathcal{F}_b$ **do**
    $C_L = C(Q(\bar{f}_i, f_j)) + C(Q(\bar{f}_i, \bar{f}_j))$
    $C_R = C(Q(f_i, f_j)) + C(Q(f_i, \bar{f}_j))$
    if $BestL.C(f_i) > C_L$ **then**
      $BestL.C(f_i) \leftarrow C_L$
    if $BestR.C(f_i) > C_R$ **then**
      $BestR.C(f_i) \leftarrow C_R$
return $\min_{f_i \in \mathcal{F}_b} BestL.C(f_i) + BestR.C(f_i)$

the sum of per-instance cost for a dataset $\mathcal{D}$:

$$P(\mathcal{D}) = \sum_{(b,y)\in\mathcal{D}} (y, y^2, 1)\,. \tag{6.5}$$

The resulting sums of $\sum y$, $\sum y^2$ and $n = |\mathcal{D}|$ can be used to obtain the SSE through Eq. (6.4).

$$C(\sum y, \sum y^2, n) = \sum y^2 - \frac{(\sum y)^2}{n}\,. \tag{6.6}$$

The optimal label $\hat{y} = \bar{y}$ is obtained through $\sum y/n$.

This break-down in per-instance costs allows for a similar performance gain as obtained by Demirović et al. (2022) by precomputing $Q(f_i) = P(\mathcal{D}_{f_i})$ and $Q(f_i, f_j) = P(\mathcal{D}_{f_i,f_j})$. Based on these precomputations, the values for other splits, such as $P(\mathcal{D}_{\bar{f}_i,f_j}) = Q(\bar{f}_i, f_j)$ can be obtained as follows:

$$\begin{aligned} Q(\bar{f}_i) &= P(\mathcal{D}) - Q(f_i)\,, \\ Q(f_i, \bar{f}_j) &= Q(f_i) - Q(f_i, f_j)\,, \\ Q(\bar{f}_i, f_j) &= Q(f_j) - Q(f_i, f_j)\,, \\ Q(\bar{f}_i, \bar{f}_j) &= P(\mathcal{D}) - Q(f_i) - Q(f_j) + Q(f_i, f_j)\,. \end{aligned} \tag{6.7}$$

Fig. 6.2 and Algorithm 9 show how the values for $Q(f_i)$ and $Q(f_i, f_j)$ can be efficiently computed by looping only over the features present for every instance. After precomputing the values for $P(\mathcal{D}), Q(f_i)$, and $Q(f_i, f_j)$, the algorithm loops

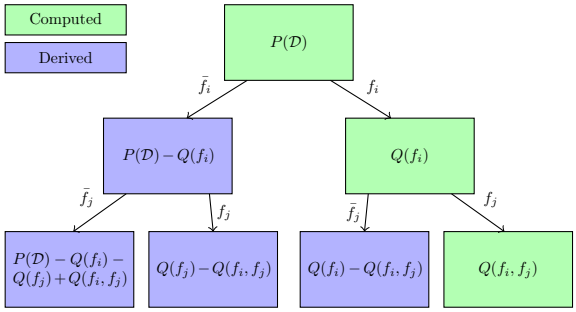


Figure 6.2: For depth-two trees, only the green values need to be precomputed per feature pair f_i and f_j. This can be done efficiently by looping only over the list of satisfied features per instance in $\mathcal{D}$. The other (blue) values can be derived according to Eq. (6.7).

over all possible combinations of two feature splits, computes the costs of the four resulting leaf nodes, and for each first-level split f_i, stores the costs of the best left and right subtree. Finally, it finds the first-level split that minimizes the total cost.

This procedure reduces the runtime for finding depth-two trees from $O(|\mathcal{F}|^2|\mathcal{D}|)$ to $O(m^2|\mathcal{D}|)$, with m the maximum number of positive features in any instance. Since binary features are commonly sparse, this results in a significant decrease in runtime at the cost of $O(|\mathcal{F}|^2)$ extra memory.

Lower bounds Scalability is further increased by lower bounds that help prune the search. We incorporate the equivalent points, k-means, and look-ahead lower bound by Zhang et al. (2023), and the similarity lower bound by Demirović et al. (2022) by using the worst-case contribution of one instance by Dunn (2018). We implement the equivalent points lower bound by grouping instances with equivalent feature vectors b as one instance with a weight, label y, and y^2 equal to the sum of the combined instances.

6.5. Piecewise Linear Regression Trees

Instead of a constant predictor in each leaf node, this section explains how optimal trees with linear regression models in each leaf node can be computed using DP. We consider both multiple linear and simple linear regression models.

6.5.1. Multiple Linear Regression

For multiple linear regression, we optimize an elastic net regression model in every leaf node. An elastic net promotes model sparsity by penalizing the L1-norm of the coefficients $\hat{\beta}$ with a factor κ and the L2-norm with a factor γ. We reuse the DP formulation of Eq. (6.3), but replace the base case ($d = 0$) with the following:

$$\min_{\hat{\beta}_0, \hat{\beta}} \sum_{(x,b,y)\in\mathcal{D}} (y - \hat{\beta}_0 - x^T\hat{\beta})^2 + \kappa\|\hat{\beta}\|_1 + \gamma\|\hat{\beta}\|_2^2 \,. \tag{6.8}$$

We use the GLMNet coordinate descent algorithm (Friedman et al., 2010) to compute the elastic net model. With GLMNet, standardization of the data takes $O(|\mathcal{D}||\mathcal{F}|)$ steps. Each iteration in the coordinate descent takes $O(m|\mathcal{F}|)$ steps, where m is the number of nonzero coefficients in the model. Adding a new nonzero coefficient takes $O(|\mathcal{D}||\mathcal{F}|)$ steps. We stop the computation after 10,000 iterations or when the change in SSE is less than 0.1%.

6.5.2. Simple Linear Regression

It remains an open challenge to formulate an efficient per-instance cost breakdown for multiple linear regression, and therefore we do not provide a special algorithm for trees of depth two. Instead, we consider the special case of *simple* linear regression. Simple linear regression considers only a single explanatory variable. This makes it easier to compute and also more human-comprehensible because the model can easily be plotted in 2D. The leaf node function becomes

$$\min_{j,\hat{\beta}_0,\hat{\beta}_j} \sum_{(x,b,y)\in\mathcal{D}} (y - \hat{\beta}_0 - x_j\hat{\beta}_j)^2 + \gamma\hat{\beta}_j^2 \,. \tag{6.9}$$

In this case, we only use the L2-norm (ridge) penalization, since the main aim for the L1 penalization is to reduce the number of non-zero coefficients, but simple linear regression already considers only one non-zero coefficient.

Per-instance costs In Appendix 6.B, we derive the formulas for an optimal simple linear regression model, which we here rewrite in terms of sums that we can precompute:

$$\hat{\beta}_0 = \sum y/n - \hat{\beta}_j \sum x_j/n \,, \tag{6.10}$$

$$\hat{\beta}_j = \frac{n\sum x_j y - \sum y \sum x_j}{n\sum x_j^2 - (\sum x_j)^2 + n\gamma} \,. \tag{6.11}$$

By expanding Eq. (6.9), and rewriting it in terms of computable sums, it can be seen that the SSE of a model with intercept $\hat{\beta}_0$ and slope $\hat{\beta}_j$ is

$$\sum y^2 - 2\hat{\beta}_j \sum y x_j - 2\hat{\beta}_0 \sum y + \hat{\beta}_j^2 \sum x_j^2 + 2\hat{\beta}_0\hat{\beta}_j \sum x_j + \gamma\hat{\beta}_j^2 + |\mathcal{D}|\hat{\beta}_0 \,. \tag{6.12}$$

The sums necessary for computing Eqs. (6.10)-(6.12), can be precomputed as in the piecewise constant case:

$$P(\mathcal{D}) = \textstyle\sum_{(x,b,y)\in\mathcal{D}} (y, y^2, 1, x_1, x_1^2, x_1 y, ..., x_{|\mathcal{F}|}, x_{|\mathcal{F}|}^2, x_{|\mathcal{F}|} y) \,. \tag{6.13}$$

The breakdown of the per-instance costs of simple linear regression is more complex than in the constant case. Before, only the sum of y, the sum of y^2, and the number of instances were computed, but now also for every continuous feature f the sum of x_f, x_f^2, and $x_f y$ is computed, resulting in a tuple of size $3+3|\mathcal{F}|$. With this new definition of $P(\mathcal{D})$, we again apply Eq. (6.7) to indirectly compute the sums for all possible leaf nodes in a depth-two tree. With all these sums precomputed as before using the

Method	Type	Reference
Piecewise constant regression trees		
CART	Greedy	Breiman et al. (1984)
GUIDE	Greedy	Loh (2002)
IAI	Local search	Dunn (2018)
ORT	Optimal MIP	Dunn (2018)
DTIP	Optimal MIP	Verwer and Zhang (2017)
OSRT	Optimal DP	Zhang et al. (2023)
SRT-C	Optimal DP	This chapter
Piecewise simple linear regression trees		
GUIDE-SL	Greedy	Loh (2002)
SRT-SL	Optimal DP	This chapter
Piecewise linear regression trees		
GUIDE-L	Greedy	Loh (2002)
IAI-L	Local search	Dunn (2018)
ORT-L	Optimal MIP	Dunn (2018)
SRT-L	Optimal DP	This chapter

Table 6.1: Overview of the analyzed methods. The DP methods split on the binarized data. The piecewise linear methods use the numerical data for training the linear models in the leaf nodes.

technique from Algorithm 9, we can compute the SSE $C_j(.)$ for when feature j is used as the explanatory variable using Eqs. (6.10)-(6.12). We do this for each feature and select the linear model with the smallest SSE: $C(\sum y, ...) = \min_j C_j(\sum y, ...)$. Algorithm 9 can now be applied in the same way as before and again yields a large scalability improvement.

6.6. Experiments

In our experiments, we first provide a scalability analysis and measure the effect of the depth-two algorithms. Second, we compare out-of-sample performance with the state of the art. The results show that our methods outperform the state-of-the-art optimal methods in scalability by one or more orders of magnitude and that the depth-two algorithm improves scalability by one order of magnitude on average. The out-of-sample analysis shows that our methods have similar out-of-sample performance as the state of the art.

6.6.1. Experiment Setup

Data Table 6.2 lists the benchmarking datasets. All datasets were obtained from the UCI Repository (Dua and Graff, 2017) and split into five folds. Categorical variables are binarized using one-hot encoding. Numerical features are binarized by training a regression tree for only that feature with at most 10 branching nodes. The splits on each branching node are used as binary predicates. The DP methods use the binarized features as predicates for the tree splits. The other methods use

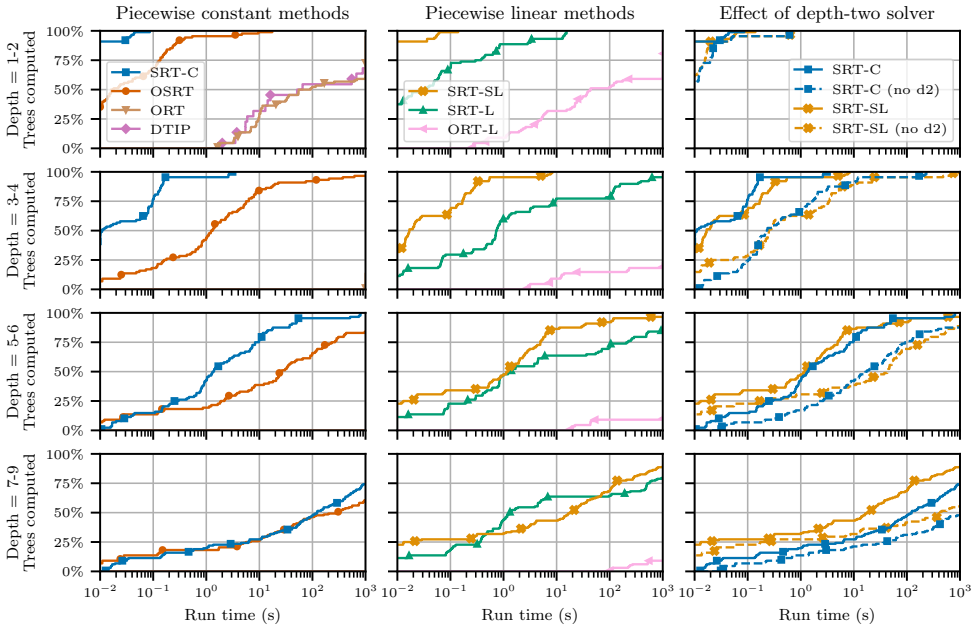


Figure 6.3: Runtime comparison for optimal methods for $d = 1...9$. Each line shows for what percentage of problem instances the optimal tree could be computed within the given runtime (higher is better). The right-most plot shows the effect of our depth-two algorithms. Our methods outperform previous optimal methods by one or more orders of magnitude. Note the logarithmic scale on the horizontal axis.

the original numerical data. All piecewise linear regression methods compute their linear models on the numerical data.

Hardware Experiments were run with one thread on an Intel Xeon E5-6248R 3.0GHz with 100GB RAM. All experiments were run with a time-out of 15 minutes.

Methods We have implemented our methods in C++ using the STreeD framework (Van der Linden et al., 2023) and provide it as a Python package.[1] We name our STreeD regression tree methods SRT-C, SRT-SL, and SRT-L, referring to our piecewise constant, simple linear, and multiple linear regression methods respectively.[2]

Table 6.1 lists all methods included in our experiments. From Bertsimas and Dunn (2017), Bertsimas et al. (2017), and Dunn (2018), we include the univariate optimal MIP methods for both piecewise constant and linear regression (ORT and ORT-L). In the same work, Dunn (2018) also proposes a local search method for both cases, which confusingly are also named ORT and ORT-L. Instead, we refer to these methods as IAI and IAI-L after their implementation by Interpretable AI (2023). For the piecewise linear methods, we adhere to the rule of thumb that the number of training samples for a linear model should be at least 10 times the number

[1] `https://github.com/AlgTUDelft/pystreed`
[2] See `https://github.com/mimvdb/regression-murtree` for our experiment setup.

of independent variables, thus requiring each leaf node to have at least $10|\mathcal{F}|$, or 10 instances in the case of simple linear regression. The MIP methods optimize the mean absolute error and are solved with Gurobi 9.5.

6.6.2. Scalability

Comparison with optimal methods To compare the scalability of optimal methods, we run each optimal method on each full dataset for depth $d = 1...9$ and for normalized $\lambda = 0.1$, 0.01, 0.001, and 0.0001 (except the much larger Household dataset, which we use below to measure scalability with respect to dataset size).

Fig. 6.3 shows a cumulative distribution plot of the runtime required for finding the optimal solution, divided in rows for different depth limits. It shows how our DP approach outperforms the MIP approaches by several orders of magnitude, both for piecewise constant and linear trees. The MIP methods ORT and DTIP do not scale beyond depth two and ORT-L only scales for larger depth for the smallest datasets. It also shows that SRT-C on average is 18 times faster than OSRT (geometric mean performance ratio). The difference is smaller for larger depth limits because then the effect of the depth-two solver for SRT-C becomes smaller. The greatest difference is observed for computing trees of depth three: SRT-C is on average 131 times faster than OSRT.

Comparison with heuristics The heuristics CART and GUIDE(-L) easily outperform all optimal methods in scalability and are therefore not included in our scalability analysis. IAI was not included in Fig. 6.3 because it is not an optimal method and its license requirement prevented us from running it on our distributed experiment server setup. Instead, we compared IAI with SRT-C separately on a local machine with an Intel i7-6600U CPU with 8GB RAM.

On average, SRT-C is faster than IAI for $d = 3$ and $d = 4$ by 26 and 5 times, respectively. For $d = 6$, IAI is 9 times faster. This is expected, as IAI is a local search method and SRT-C an optimal method. SRT-L and IAI-L show a similar relationship, but not as pronounced. SRT-L is faster than IAI-L up to $d = 3$ and has similar performance for $d = 4$ and $d = 5$. At $d = 6$, IAI-L is approximately 2 times faster than SRT-SL. Though SRT-C and SRT-L scale exponentially with the number of features and maximum depth and IAI and IAI-L scale linearly for the number of features and depth, both SRT-C and SRT-L remain competitive in runtime with these local search algorithms for the cases we considered, while guaranteeing optimal solutions.

Scalability break down Fig. 6.4 shows that, as expected, all DP methods scale linearly for the number of instances and exponentially for the number of features.

Surprisingly, the performance of SRT-SL is very close to SRT-C's performance and much better than OSRT's, which means we can now fit a simple linear regression model in each leaf node even faster than previous methods could fit constant predictors in each leaf node.

Even SRT-L, which fits a full linear regression model in its leaves, scales similarly to OSRT (with constant predictors) for an increasing number of features. However, because of the coordinate descent algorithm, it scales slightly worse for an increasing

Dataset	$\|\mathcal{D}\|$	$\|\mathcal{F}\|$	$\|\mathcal{F}_b\|$	Heuristic			Optimal DP	
				CART	GUIDE	IAI	OSRT	SRT-C
Airfoil	1503	5	28	0.61	0.59	**0.69**	-	0.67
Auction	2043	7	26	0.95	0.89	**0.97**	**0.97**	**0.97**
Auto MPG	392	7	38	0.77	0.80	0.80	-	**0.81**
Energy (C)	768	8	37	0.96	0.95	0.96	**0.97**	**0.97**
Energy (H)	768	8	37	0.99	0.97	0.99	**1.00**	**1.00**
Household	2049280	3	26	**1.00**	-	-	-	0.98
Optical Net.	640	7	42	**0.96**	0.91	0.93	0.95	0.95
Real Estate	414	6	46	**0.66**	0.62	0.63	-	**0.66**
Seoul Bike	8760	9	57	0.72	0.62	**0.74**	-	-
Servo	167	2	17	0.78	-0.02	0.78	**0.89**	**0.89**
Synch.	557	4	34	**1.00**	**1.00**	**1.00**	0.99	0.99
Yacht	308	6	43	**0.99**	**0.99**	**0.99**	-	**0.99**
Best				5	2	5	4	**7**

Table 6.2: Out-of-sample R^2 results for the piecewise constant methods with maximum depth $d = 5$. Time-outs indicated by '-'. Best results are marked bold. $|\mathcal{D}|$ is the number of instances, $|\mathcal{F}|$ the number of original features and $|\mathcal{F}_b|$ is the number of binary features.

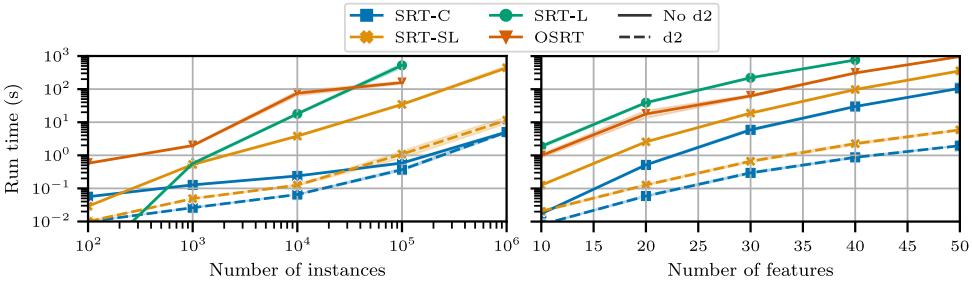


Figure 6.4: Runtime increase for the number of instances (Household dataset) and features (Seoul Bike dataset) for $d = 4$. OSRT exceeds the memory limit for $|\mathcal{D}| \geq 10^6$.

number of instances. OSRT exceeds the memory limit of 100GB for $|\mathcal{D}| \geq 10^6$.

Effect of the depth-two algorithms Fig. 6.3 and Fig. 6.4 also show the effect of our depth-two algorithm. For SRT-C and SRT-SL, on average, it improves the runtime by a factor 8 and 10 respectively. For trees of depth three, it improves the runtime of SRT-C and SRT-SL on average by a factor 20 and 12 respectively. At larger depth, the dataset splits are much smaller and thus the effect of the depth-two solver is less significant. For SRT-SL, the performance improvement remains significant even at larger depth limits. The benefit of the depth-two algorithm increases with more features. SRT-C sees a diminishing impact of the depth-two algorithm for an increasing number of instances, but for SRT-SL its impact is even greater.

		Simple Regression		Multiple Regression			
						SRT-L*	
Dataset	LR	GUIDE-SL	SRT-SL*	GUIDE-L	IAI-L	Lasso	Elastic Net
Airfoil	0.51	0.65	0.72	0.85	0.88	**0.89**	-
Auction	0.38	0.89	**0.96**	0.94	0.94	0.94	0.94
Auto MPG	0.82	0.83	**0.84**	**0.84**	**0.84**	**0.84**	**0.84**
Energy (C)	0.89	0.95	**0.97**	**0.97**	**0.97**	**0.97**	**0.97**
Energy (H)	0.92	0.98	**1.00**	0.99	0.99	0.99	0.99
Household	**1.00**	-	**1.00**	-	-	-	-
Optical Net.	0.29	0.15	**0.96**	0.20	0.59	0.73	0.77
Real Estate	0.58	0.59	0.58	0.62	**0.63**	**0.63**	**0.63**
Seoul Bike	0.56	0.69	-	**0.79**	-	-	-
Servo	0.45	-0.05	**0.64**	-0.03	0.50	0.49	0.53
Synch.	**1.00**	0.96	**1.00**	0.95	**1.00**	**1.00**	**1.00**
Yacht	0.63	**0.99**	**0.99**	0.98	0.98	**0.99**	**0.99**
Best per cat.		3	**10**	5	7	8	**9**
Best overall	2	1	**9**	3	4	6	5

Table 6.3: Out-of-sample R^2 results for the piecewise linear methods with $d = 4$. Time-outs indicated by '-'. The best results are marked in bold. Optimal methods are marked with an *.

6.6.3. Out-of-Sample Performance

Setup For out-of-sample analysis, each method is tested on each of the five folds of the datasets with the other folds as training data. For CART, OSRT, SRT-C, SRT-SL, and SRT-L we hyper-tune the regularization parameter λ using five-fold cross-validation. For GUIDE, GUIDE-SL, GUIDE-L, IAI, and IAI-L, we use the default hyper-tuning as included in these methods. Furthermore, SRT-L and IAI-L tune the Lasso and Ridge penalization (if applicable) in a second hyperparameter-tuning phase, as explained by Dunn (2018). The piecewise constant and linear methods are trained with a maximum depth of five and four respectively. The MIP models ORT, ORT-L, and DTIP resulted in time-outs for almost every scenario even without hyper-tuning, and are therefore left out of the results. We show training scores at time-out (without hyper-tuning) for the MIP methods and all other methods in Appendix 6.C. Every method, except LR, CART, SRT-C, and SRT-SL resulted in time-outs or out-of-memory errors for the Household dataset.

Results Table 6.2 shows that the out-of-sample coefficients of determination (R^2) for all piecewise constant methods are close, with each method scoring best for at least one dataset. SRT-C most often performs best. With a 5% significance level, Wilcoxon signed rank tests show that SRT-C performs better than GUIDE and equal to IAI, ORST, and CART. OSRT results in time-outs for several datasets.

Table 6.3 shows the out-of-sample R^2-scores for all piecewise linear methods. For comparison, the results of linear regression are also included. Since IAI-L uses lasso penalization, we compare it with both SRT-L using only lasso regularization and

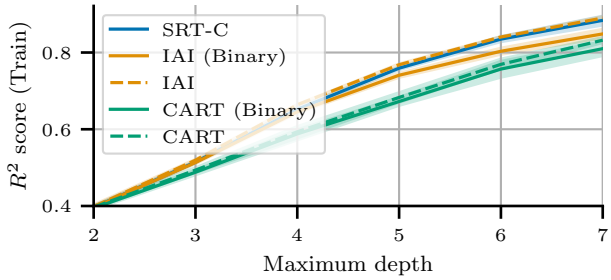


Figure 6.5: Training R^2 score for Airfoil. SRT-C (with binary features) performs similarly to IAI with original features, better than IAI with binary features, and better than CART in both cases

also elastic net regularization.

All methods perform significantly better than standard linear regression. SRT-SL, with its simple linear regression models, performs best for most datasets. The runner-up is SRT-L (with either regularization technique). Both SRT-SL and SRT-L perform significantly better than GUIDE-SL and GUIDE-L. The other differences are not significant.

Binarization and optimality To show the effect of preprocessing the feature data for possible binary tree splits and the difference between optimal and non-optimal methods, we also compare the training R^2 score of SRT-C with that of IAI and CART when the latter are trained with the original numerical and with the binarized data. Fig. 6.5 shows that SRT-C, despite losing information in the binarization, obtains approximately the same training R^2 score as IAI and outperforms IAI and CART clearly when those methods are also trained with binary features.

Discussion Our analysis shows that our optimal methods have better out-of-sample generalization than previous heuristics, although the differences are small. The optimal methods guarantee optimality on the training data for a given size limit and binarization. However, the choice of binarization and tree-size hypertuning techniques have not been studied well enough yet in the literature, which prevents full utilization of the strength of optimal methods.

Though our method SRT-C obtains the same optimal solutions as OSRT by Zhang et al. (2023), our conclusion is more cautious than theirs, when they conclude that OSRT outperforms CART, GUIDE, and IAI in out-of-sample performance. Differences that may explain this are: (1) they compare all methods on the binarized data, whereas we train methods with the original numerical data if possible; (2) for most datasets, they binarize numerical features into four binary features, instead of at most 10, as we do; (3) their setup for GUIDE mistakenly reuses the train predictions to assess the test performance, which explains why they report a poor performance for GUIDE; and (4) they do not test the statistical significance of their results.

6.7. Conclusion

We present three new dynamic programming (DP) approaches for regression trees: a new algorithm for computing piecewise constant regression trees that improves scalability using a special algorithm for depth-two trees; and the first optimal DP methods for piecewise multiple and simple linear regression trees. For piecewise simple linear regression trees, we also provide a special algorithm for depth-two trees. The results show that our methods improve scalability by one or more orders of magnitude in comparison to the state of the art, and mainly so because of our depth-two algorithms. The out-of-sample performance of our methods is on par with or better than the state of the art.

Complexity-tuning techniques should be further researched to fully exploit the power of optimal regression trees. Other extensions include dealing with non-binary features for the branching decisions, as done for example by Mazumder et al. (2022), and further exploiting the subtree independence through parallelization of the algorithm.

Appendices for Chapter 6

6.A. Pseudocode

Algorithm 10 shows the full pseudocode for SRT. It first checks if the dataset size is at least the minimum leaf node size. If the depth budget d or node budget n is zero, it returns the SSE of a leaf node using either Eq. (6.4), (6.8), or (6.9), for SRT-C, SRT-SL, and SRT-L respectively. If the depth or node budget exceeds what is possible according to the other budget, the budgets are updated and SRT is called again. If the regularization costs λ multiplied by the node budget exceeds the upper bound, the node budget is updated. SRT then checks the cache to see if the subproblem has been encountered before, and if so, returns it. If the depth budget is at most two, SRT uses the special depth-two algorithm as explained in Algorithm 9 (not for SRT-L).

In all other cases, SRT will consider all possible branching decisions and node budget divisions. If a split does not respect the minimum leaf node size, it is skipped. Otherwise, lower bounds are obtained (from cache or using the lower bounds specified in the main body of this chapter). If the lower bound for a split exceeds the upper bound, the split is skipped. Otherwise, a left and right subtree are generated, while updating the upper bound by subtracting the lower bound and the regularization parameter from the current upper bound. If the combination of the two solutions yields a better solution, the current best solution is replaced. Finally, if a solution below the upper bound has been found, it is stored in the cache. Otherwise, the upper bound is stored as a lower bound in the cache.

6.B. Simple Linear Regression Derivation

The error for simple linear regression, including the Ridge L2-norm penalization, is as follows:

$$\text{SSE} = \sum_{i=1}^{n}(y_i - \hat{\beta}_0 - \hat{\beta}x_i)^2 + \gamma\hat{\beta}^2 \tag{6.14}$$

The optimal value for the intercept $\hat{\beta}_0$ is found by setting the derivative to zero:

$$\frac{\partial \text{SSE}}{\partial \hat{\beta}_0} = \sum_{i=1}^{n}(-2y_i + 2\hat{\beta}_0 + 2\hat{\beta}x_i) \tag{6.15}$$

$$0 = \sum_{i=1}^{n}(-y_i + \hat{\beta}_0 + \hat{\beta}x_i) \tag{6.16}$$

$$\hat{\beta}_0 = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{\beta}x_i) \tag{6.17}$$

$$\hat{\beta}_0 = \bar{y} - \hat{\beta}\bar{x} \tag{6.18}$$

Algorithm 10: Pseudo-code for SRT given a dataset $\mathcal{D}$, feature sets $\mathcal{F}$ and $\mathcal{F}_b$, a depth budget d, a node budget n, an upper bound ub, a complexity cost λ and a minimum leaf node size M.

```
SRT(D, d, n, ub)
    if |D| < M then return ∞
    if d = 0 ∨ n = 0 then
        s ← solveLeaf(D)
        if s ≥ ub then return ∞
        return s
    if n > 2^d − 1 then return SRT(D, d, 2^d − 1, ub)
    if d > n then return SRT(D, d, d, ub)
    if nλ > ub then return SRT(D, d, ⌊ub/λ⌋)
    ⟨s, lb, stat⟩ ← cache[D, d, n]
    if lb ≥ ub then return ∞
    if stat = optimal then return s
    if d ≤ 2 then s ← SolveD2(D, d, n)
    else
        s ← solveLeaf(D)
        if s ≥ ub then s ← ∞
        for f ∈ F_b, n_L ∈ [0, n − 1] do
            if |D_f| < M ∨ |D_f̄| < M then continue
            n_R ← n − n_L − 1
            lb_L ← LB(D_f̄, d − 1, n_L)
            lb_R ← LB(D_f, d − 1, n_R)
            lb = lb_L + lb_R + λ
            if lb ≥ ub then continue
            ub_L ← ub − lb_R − λ
            s_L ← SRT(D_f̄, d − 1, n_L)
            if s_L = ∞ then continue
            ub_R ← ub − s_L − λ
            s_R ← SRT(D_f, d − 1, n_R)
            if s_R = ∞ then continue
            if s_L + s_R + λ < s then s ← s_L + s_R + λ
    if s ≠ ∞ then cache[D, d, n] ← ⟨s, s, optimal⟩
    else cache[D, d, n] ← ⟨∞, ub, lowerbound⟩
    return s
```

The optimal value for the slope $\hat{\beta}$ is also found by setting the derivative to zero, and by filling in Eq. (6.18) for $\hat{\beta}_0$:

$$\frac{\partial \, \text{SSE}}{\partial \hat{\beta}} = \sum_{i=1}^{n} (-2x_i y_i + 2\hat{\beta}_0 x_i + 2\hat{\beta} x_i^2) + 2\gamma\hat{\beta} \tag{6.19}$$

	Heuristic			Optimal MIP		Optimal DP	
Dataset	CART	GUIDE	IAI	DTIP	ORT	OSRT	SRT-C
Airfoil	0.68	0.67	**0.77**	0.40 *	0.56 *	0.76	0.76
Auction	0.96	0.91	**0.97**	0.14 *	-0.49	**0.97**	**0.97**
Auto MPG	0.93	0.91	**0.94**	0.91 *	0.88 *	**0.94**	**0.94**
Energy (C)	0.96	0.96	0.97	0.94 *	0.93 *	**0.98**	**0.98**
Energy (H)	0.99	0.98	0.99	0.97 *	0.96 *	**1.00**	**1.00**
Household	**1.00**	OoM	**1.00** *	OoM	OoM	OoM	0.98
Optical Net.	**1.00**	0.80	**1.00**	**1.00** *	0.96 *	**1.00**	**1.00**
Real Estate	0.86	0.76	**0.89**	0.80 *	0.78 *	0.88	0.88
Seoul Bike	0.73	0.63	0.75	-0.08 *	-1.19	0.24 *	**0.76**
Servo	0.96	0.12	**0.99**	0.98 *	0.93 *	**0.99**	**0.99**
Synch.	**1.00**	**1.00**	**1.00**	0.99 *	0.99 *	0.99	0.99
Yacht	**1.00**	**1.00**	**1.00**	**1.00** *	0.99 *	**1.00**	**1.00**
Best	4	2	9	2	0	7	8

Table 6.4: Training R^2 scores (without hyper-tuning) for trees with constant predictors at $d = 5$. An * indicates this is the best result obtained at the time-out (900s). The best results per category are shown in bold. OoM indicates out of memory.

$$0 = \sum_{i=1}^{n}(-x_i y_i + \hat{\beta}_0 x_i + \hat{\beta} x_i^2) + \gamma\hat{\beta} \tag{6.20}$$

$$= \sum_{i=1}^{n}\left(-x_i y_i + \left(\bar{y} - \hat{\beta}\bar{x}\right) x_i + \hat{\beta} x_i^2\right) + \gamma\hat{\beta} \tag{6.21}$$

$$\hat{\beta} = \frac{\sum_{i=1}^{n}(x_i y_i - \bar{y} x_i)}{\sum_{i=1}^{n}(x_i^2 - \bar{x} x_i) + \gamma} \tag{6.22}$$

By observing that $\bar{y} = \sum_i y_i/n$ and $\bar{x} = \sum_i x_i/n$, we can rewrite to Eq. (6.10) and (6.11):

$$\hat{\beta}_0 = \sum_{i=1}^{n} y_i/n + \hat{\beta}\sum_{i=1}^{n} x_i/n \tag{6.23}$$

$$\hat{\beta} = \frac{n\sum_{i=1}^{n} x_i y_i - \sum_{i=1}^{n} y_i \sum_{i=1}^{n} x_i}{n\sum_{i=1}^{n} x_i^2 - (\sum_{i=1}^{n} x_i)^2 + n\gamma} \tag{6.24}$$

We set the ridge penalty $\gamma = \sigma^2\gamma'$, with σ^2 the variance of the feature vector x, and γ' a hyper-parameter. We tune for $\gamma' \in \{0, 0.01, 0.1, 1, 10, 100, 1000\}$.

6.C. Training Results

Tables 6.4 and 6.5 show the training R^2-scores for all methods. We train each method on the five folds of each dataset and report the average training score. When at least one run resulted in a time-out ($> 900s$), we mark it with an asterisk and report the training score at time-out.

	Simple linear regression		Multiple linear regression			
	Heuristic	Opt. DP	Heuristic		Opt. MIP	Opt. DP
Dataset	GUIDE-SL	SRT-SL	GUIDE-L	IAI-L	ORT-L	SRT-L
Airfoil	0.69	**0.80**	0.86	**0.92**	0.79 *	**0.92**
Auction	0.91	**0.96**	0.94	**0.95**	-0.50	**0.95**
Auto MPG	0.91	**0.93**	0.87	**0.90**	0.89 *	**0.90**
Energy (C)	0.96	**0.98**	**0.97**	**0.97**	**0.97** *	**0.97**
Energy (H)	0.98	**1.00**	**0.99**	**0.99**	0.98 *	**0.99**
Household	OoM	**1.00**	OoM	**1.00** *	OoM	**1.00** *
Optical Net.	0.68	**0.98**	0.27	0.11	-0.01	**0.90**
Real Estate	0.76	**0.85**	0.68	**0.75**	0.70 *	**0.75**
Seoul Bike	0.69	**0.76**	**0.80**	0.77	-0.97 *	0.76 *
Servo	0.10	**0.87**	0.04	0.75	0.34	**0.76**
Synch.	0.97	**1.00**	0.95	**1.00**	**1.00**	**1.00**
Yacht	0.99	**1.00**	**0.99**	0.91	0.91 *	**0.99**
Best per cat.	0	12	4	8	2	11

Table 6.5: Training R^2 scores (without hyper-tuning) for trees with linear predictors at $d = 4$. An * indicates this is the best result obtained at the time-out (900s). The best results per category are shown in bold. OoM indicates out of memory.

MIP results The results show that (even without hyper-tuning) DTIP never finishes before the time-out and ORT and ORT-L only twice and four times respectively. Furthermore, they never exceed the training score of SRT-C and SRT-L.

To keep the objective linear, we train DTIP, ORT, and ORT-L by minimizing the mean absolute error. We have also tested minimizing the quadratic mean squared error, but this yielded even larger run times. This difference can explain why ORT, when it obtains the optimal solution, is still worse than SRT-C.

Because the MIP methods time out in almost every case even without hyper-tuning, we did not include these methods in our evaluation in the main text.

GUIDE is a greedy algorithm like CART but differs in a few ways: it creates unbiased splits, it can group multiple values for categorical variables in one branch, and it uses significance tests for each split. These significance tests prevent overfitting and also explain why GUIDE's training accuracy is typically lower than CART's in Table 6.4. GUIDE-SL is always worse than SRT-SL in Table 6.5. GUIDE-L is best for one dataset, and similar or worse for all others.

IAI IAI and IAI-L do not provide a time limit parameter. Therefore, when these methods exceeded the time limit for the household dataset, we reran it with a reduced number of random restarts until the observed run time was only just above the time limit. For the household dataset, we ran IAI with 18 random restarts and IAI-L with 15 random restarts. For the other datasets, we use the default 100 random restarts.

The training performance of IAI and SRT-C are close. On four datasets, IAI's R^2 is marginally better because it uses the numerical feature data directly instead of the binarized features. On three datasets, SRT-C is marginally better than IAI

because its exhaustive search considers trees that IAI's local search does not find.

Similarly, for multiple linear regression, the performance is close. IAI-L is marginally better for one dataset. SRT-L is marginally better for one dataset and significantly better for two others.

OSRT The two DP methods OSRT and SRT-C obtain as expected the same training score. The only differences are for the household dataset, where OSRT runs out of memory, and the Seoul-bike dataset, where OSRT does not find the optimal solution within the time limit but SRT-C does.

7

Empirical Analysis of Optimal and Greedy Decision Trees

Preface This chapter provides an in-depth empirical analysis of the optimal and greedy top-down induction approach to decision tree learning. This analysis helps understand the differences and similarities between the two, and both motivates and provides limits on the utility of optimal decision trees.

Abstract Recently there has been a surge of interest in optimal decision tree (ODT) methods that globally optimize accuracy directly, in contrast to traditional approaches that locally optimize an impurity or information metric. However, the value of optimal methods is not well understood yet, as the literature provides conflicting results, with some demonstrating superior out-of-sample performance of ODTs over greedy approaches, while others show the opposite. Through a novel extensive experimental study, we provide new insights into the design and behavior of learning decision trees. In particular, we identify and analyze two relatively unexplored aspects of ODTs: the objective function used in training trees, and tuning techniques. Thus, we address these three questions: what objective to optimize in ODTs; how to tune ODTs; and how do optimal and greedy methods compare? Our experimental evaluation examines 11 objective functions, six tuning methods, and six claims from the literature on optimal and greedy methods on 180 real and synthetic datasets. Through our analysis, both conceptually and experimentally, we show the effect of (non-)concave objectives in greedy and optimal approaches; we highlight the importance of proper tuning of ODTs; support and refute several claims from the literature; provide clear recommendations for researchers and practitioners on the usage of greedy and optimal methods; and code for future comparisons.

This chapter is available as a preprint paper: Van der Linden, Vos, De Weerdt, Verwer, and Demirović (2024), "Optimal or Greedy Decision Trees? Revisiting their Objectives, Tuning, and Performance", in *arXiv preprint arXiv:2409.12788*. For this chapter, the introduction has been shortened and the related work has become part of Chapter 2.

7.1. Introduction

While *greedy* decision tree learning has been extensively studied given its long history, *optimal* decision tree research (ODT) is a much younger field, with the last decade seeing major advancements. However, so far, research for ODTs has been mostly limited to improving *scalability* by reducing runtimes and supporting larger datasets.

Given that the main technical innovations for optimal methods are relatively recent, unlike in the field of greedy heuristics, principled ways of using optimal decision trees for out-of-sample performance have been comparatively under-explored. For example, the field of greedy decision trees shows a large variety of splitting criteria (Mingers, 1989b; Buntine and Niblett, 1992; Shih, 1999; Raileanu and Stoffel, 2004; Wang and Xia, 2017) and pruning techniques to avoid overfitting (Mingers, 1989a; Esposito et al., 1997; Patil et al., 2010). In contrast, optimal decision trees are almost exclusively trained by maximizing accuracy, possibly additionally penalizing the number of nodes (the *sparse* objective). Tuning, if done at all, is performed in different ad hoc manners, e.g., tuning the number of nodes or depth of the tree. Practices differ from paper to paper, which hinders direct comparisons.

Moreover, early comparisons between optimal and greedy approaches were limited in scope and contained claims and hypotheses that we can now refute (Section 7.4). Murthy and Salzberg (1995) lacked a scalable ODT method and therefore confined their analysis on synthetic data. Bertsimas and Dunn (2017) trained ODTs using MIP, but lack of scalability constrained most of their analysis to datasets of only 100 instances or trees with a maximum depth of two. For larger problems, their approach did not converge to optimality; therefore, the support for several of their claims remained uncertain.

Though these and other studies (Lin et al., 2020; Demirović et al., 2022) report an average improvement of the out-of-sample performance versus greedy heuristics, others have criticized ODTs for overfitting (Dietterich, 1995), observed worse results for ODTs compared to greedy heuristics (Zharmagambetov et al., 2021; Marton et al., 2024), and questioned the adjective 'optimal' (Sullivan et al., 2024). These contradictory findings illustrate the need for a more thorough understanding of the concept of optimal decision trees.

This motivated us to conduct a thorough experimental evaluation of existing decision tree methods, both greedy and optimal, focusing on objective functions used during training and different tuning approaches. Due to recent algorithmic advancements and an increase in computation power, we can now use our dynamic programming approach (Chapter 4) to surpass the scalability limitations of previous studies and analyze datasets with up to hundreds of thousands of samples. This enables us to conduct the largest evaluation to this date concerning optimal and greedy decision tree methods, taking into account 11 different objective functions, six tuning approaches, 180 real-world and synthetic datasets (small and large), and trees that go beyond small tree-depth limits. This provides us with a wealth of data to analyze and improves our understanding of how to apply greedy and optimal approaches for training decision trees.

From our new insights obtained on training ODTs, we also discuss the implications for decision tree learning in general. To keep the scope of this study manageable, we

chose to limit this study to axis-aligned binary classification trees with hard splits, which are arguably the most common type of decision trees. In more detail, we contribute the following:

- In Section 7.2, we analyze and experimentally compare nine existing greedy decision tree accuracy objectives. Since we observe that the strict concavity of these objectives, as required by greedy top-down inducting approaches (Kearns and Mansour, 1996), is counterproductive when trained to optimality, we also introduce and experiment with two new non-concave objectives. Our experiments show that greedy and optimal approaches respond oppositely to the (non-)concavity of these objectives. Additionally, we show the benefit of objectives that include a regularizing component for noisy data (in addition to the regular tree size tuning).

- In Section 7.3, we compare six complexity tuning approaches for ODTs, four of which were proposed before, and two new tuning approaches that we introduce here. Our experiments highlight the importance of tuning optimal decision tree methods, and that (surprisingly) the accuracy differences between the commonly used tuning methods are small, although there are differences in resulting tree size and runtimes.

- In Section 7.4, we analyze previous comparisons between greedy and optimal approaches, formulate best practices for future comparisons, and provide data and code to support proper benchmarking. We apply these practices in evaluating six claims from the literature on the performance of greedy and optimal trees:

  Claim 1: Optimal methods under the same depth constraint (up to depth four) find trees with 1-2% higher out-of-sample accuracy than greedy methods (Bertsimas and Dunn, 2017; Verwer and Zhang, 2017; Demirović et al., 2022).

  Claim 2: Optimal methods obtain a better accuracy-interpretability trade-off than greedy methods (Lin et al., 2020).

  Claim 3: The difference between optimal and greedy approaches diminishes with more data (Murthy and Salzberg, 1995; Costa and Pedreira, 2023).

  Claim 4: The accuracy of greedy trees remains stable when the data size increases linearly with concept complexity (Murthy and Salzberg, 1995).

  Claim 5: Optimal trees are more likely to overfit than greedy trees (Dietterich, 1995).

  Claim 6: The question length of greedy trees remains (in practice) close to that of optimal trees (Goodman and Smyth, 1988; Murthy and Salzberg, 1995).

  Our results support Claims 1, 2, and 6, and refute Claims 3, 4, and 5.

Sections 2, 3, and 4 are mostly self-contained sections, each dedicated to a single major research question as outlined above, each with its corresponding related work, technical details, experiments, and conclusions. Section 5 draws an overarching conclusion. For related work and background, we refer to Chapter 2.

7.2. The Optimization Objective for ODTs

Decision tree learning objectives typically optimize two parts: some accuracy objective (i.e., accuracy or one of its proxies, such as information gain) and a tree-complexity objective (e.g., number of nodes). In this section, we focus on the first: the accuracy objective. In Section 7.3, we discuss the tree-complexity objective.

Section 7.2.1 explains why existing greedy splitting criteria do not optimize accuracy directly. We transform these greedy criteria to ODT objectives and observe in our analysis of these objectives in Section 7.2.2, that the strict concavity traditionally required by greedy heuristics (Kearns and Mansour, 1996) is not helpful when training ODTs. Therefore, we introduce two novel non-concave objectives in Section 7.2.3. We then empirically compare all these objectives in Section 7.2.4 and discuss our findings in Section 7.2.5. Our main finding is that greedy learners perform best with strictly concave splitting criteria, whereas optimal learners achieve the best performance with non-strictly-concave objectives.

7.2.1. Greedy Proxies for Accuracy

Although the goal of decision tree learning is to maximize accuracy, TDI methods rarely optimize accuracy directly but instead optimize a proxy, such as the reduction in the χ^2-statistic in CHAID (Kass, 1980), Gini impurity by CART (Breiman et al., 1984), and information gain (entropy) by ID3 (Quinlan, 1986) and C4.5 (Quinlan, 1993b).

The reason why TDI methods do not optimize accuracy directly is that an accuracy splitting criterion is often unable to find an improving split in unbalanced data. When splitting a node, TDI methods evaluate all possible splits and choose the split that minimizes the splitting criterion value for the resulting class distributions among the new nodes of each possible split. Fig. 7.1 visually explains why using accuracy as a splitting criterion is worse at distinguishing improving splits than Gini impurity or entropy. The left side shows the function values for accuracy, Gini impurity, and entropy for binary classification. The right side shows how a locally optimal split can be found geometrically. When splitting a node with a probability of the first class of p into two new nodes with probabilities p_1 and p_2, the new weighted splitting criterion value can be found by drawing a straight line from the criterion value at point p_1 to p_2. The intersection of the straight line at p is the sum of the weighted criterion value of the two nodes. For Gini impurity and entropy, this value is always lower than the criterion value of the parent node, because both functions are *strictly concave*. Accuracy, however, is not strictly concave, and when $p \leq 0.5, p_1 \leq 0.5$, and $p_2 \leq 0.5$ (or equivalently, all are greater than or equal to 0.5), the weighted sum of the criterion value of the child nodes is the same as that of the parent node. Moreover, for any values $p_1 \leq 0.5$ and $p_2 \leq 0.5$ the weighted sum of

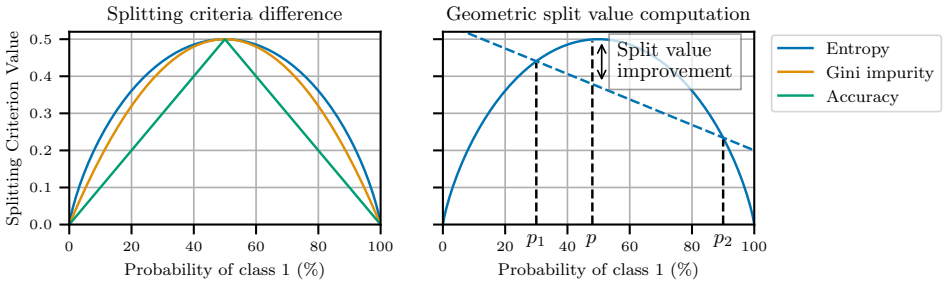


Figure 7.1: (Left) Three splitting heuristics compared. The horizontal axis shows the binary class distribution expressed as the probability of the first class, and the vertical axis shows the corresponding splitting criterion value (lower is better). (Right) Geometric interpretation of the weighted mean error of two children when p, p_1, and p_2 represent the class distributions of the parent and the two children respectively. The length of the arrow indicates the improvement in the splitting criterion value. Adapted from Flach (2012).

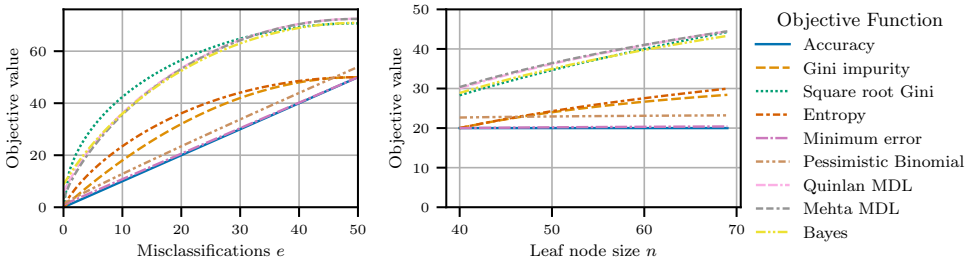


Figure 7.2: Objective values for different objective functions for a single leaf node. (Left) The leaf node size is fixed at $n = 100$. (Right) The misclassifications are fixed at $e = 20$. Surprisingly, the value of the strictly concave objectives increases for a fixed error and increasing leaf node size.

the criterion values is the same, and therefore no distinction can be made between these splits. Thus TDI heuristics require strictly concave splitting criteria (Kearns and Mansour, 1996) and therefore do not optimize accuracy directly.

7.2.2. Analysis of ODT Objectives

To increase our understanding of greedy and optimal decision tree learning approaches, we analyze accuracy and eight other existing greedy splitting and pruning criteria and rewrite them as ODT objectives: Gini impurity, square root Gini, entropy, minimum error, binomial pessimistic error (Binom.), minimum description length (MDL, two encodings: Quinlan and Mehta), and Bayesian. In Appendix 7.A, we rewrite each of these as a function f with as input the number of instances n that reach this leaf and the number of misclassifications e in this leaf. Let L be a set of leaf nodes of size n and with e misclassifications, then the minimization objective of the whole tree becomes $\sum_{(n,e)\in L} f(n, e)$.

The left of Fig. 7.2 shows the values of these objectives when the leaf node size is fixed but the number of misclassifications increases. The accuracy is a straight line

Objective	Expected (lower is better)	Observed
Gini impurity	$f(8,2) \leq f(4,2) + f(4,0)$	$f(8,2) = 3.000,\ f(4,2) + f(4,0) = 2.000$
Entropy	$f(8,1) \leq f(4,2) + f(4,0)$	$f(8,1) = 2.174,\ f(4,2) + f(4,0) = 2.000$
MDL (Quinlan)	$f(6,2) \leq f(4,2) + f(2,0)$	$f(6,2) = 4.708,\ f(4,2) + f(2,0) = 4.377$
MDL (Mehta)	$f(6,2) \leq f(4,2) + f(2,0)$	$f(6,2) = 5.513,\ f(4,2) + f(2,0) = 5.409$
Bayes	$f(6,2) \leq f(4,2) + f(2,0)$	$f(6,2) = 4.379,\ f(4,2) + f(2,0) = 4.321$

Table 7.1: Pure nodes are overvalued, resulting in splits with pure nodes (e.g., $(n, e) = (4, 0)$) and nodes that are labeled randomly $(4, 2)$, rather than keeping one node with the same misclassifications $(8, 2)$ or even less in case of entropy $(8, 1)$.

since every misclassification is counted equally. Both the pessimistic binomial score and the minimum error follow the accuracy tightly, with only a small additional cost for higher misclassifications. All other objectives follow roughly the same pattern: the first misclassifications in a node are penalized most and the additional penalty for extra misclassifications decreases.

Similarly, on the right of Fig. 7.2, when the number of misclassifications in a node is fixed but the leaf node size is changed, the accuracy is a straight line. The pessimistic binomial score and the minimum error again follow accuracy closely. Interestingly, for all other objectives, the objective value increases when the node size increases. Since lower values are preferred, this means these objectives penalize larger leaf nodes more than smaller leaf nodes with the same misclassifications.

It is counter-intuitive that the objective increases for larger nodes with a fixed error. Table 7.1 shows some examples of relative objective values that are unexpected. For example, according to the entropy criterion, it is better to have two nodes of size 4, with two misclassifications in the first node and zero in the second, than one node of size 8 with one misclassification. Entropy strongly values a pure node, even if this means a higher misclassification rate. Other objectives, such as MDL, value two nodes of size four and two with two misclassifications in the first and none in the second, more than one node of size six with also two misclassifications. Again, a small pure node is valued, even though the node of size four with two misclassifications has a high probability of being misclassified, for example, in the presence of class noise.

7.2.3. Novel Non-Concave Objectives

The odd behavior of the greedy criteria in Fig. 7.2 and Table 7.1 is a result of their strict concavity. (Strict) concavity is not a requirement for ODTs because ODTs do not consider splitting criteria and can search beyond a non-improving split. Therefore, we here introduce two *non-concave* objectives by Noel et al. (2023) that have not previously been used in decision tree learning.

M-loss: The first is the *M-loss*, here rewritten in terms of n and e:

$$f(n, e) = n \left(\frac{1}{1 - \frac{e}{n}} - 1 \right) .$$

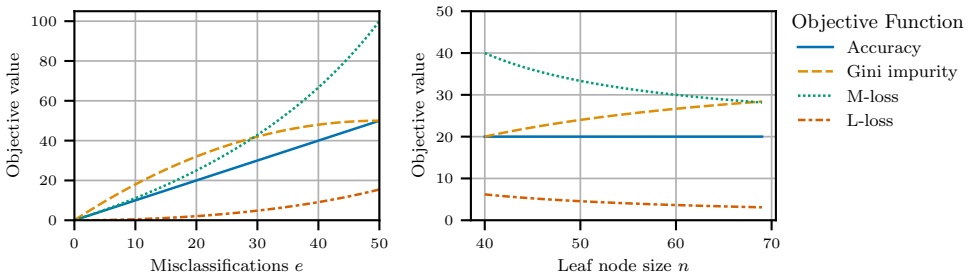


Figure 7.3: The new objectives show opposite behavior to the strictly concave objectives. Left, the leaf node size is fixed at $n = 100$. Right, the misclassifications are fixed at $e = 20$.

L-loss: The second objective function that they propose is called the *L-loss*. Rewritten in terms of n and e this becomes

$$f(n, e) = n\left(\frac{1}{\sqrt{1 - \left(\frac{e}{n}\right)^2}} - 1\right).$$

Fig. 7.3 shows the values these new functions take. For easy comparison, accuracy and Gini impurity are also included in the plot. In contrast to the strictly concave functions, the left side of Fig. 7.3 shows how the first misclassifications in a leaf node are penalized less, whereas nodes with a (close to) balanced class distribution are heavily penalized. The right side shows that increasing the leaf node size while keeping the number of misclassifications constant, decreases the penalization. Therefore, we hypothesize that these objectives obtain the desired property to penalize nodes with a close-to-equal class distribution more strongly.

7.2.4. Experiments

In our experiments we aim to answer the following questions:

1. What is the difference between the objectives on out-of-sample accuracy when trained to optimality on the training data?
2. What difference can be observed between the objectives when trained to optimality on the training data or when greedily optimized using TDI heuristics?
3. How do different objectives respond to noise and dataset size?

Experiment Setup

We empirically compare the objectives on a large benchmark set from OpenML (Vanschoren et al., 2013; Feurer et al., 2021).[1] For the sake of scalability, we selected all binary classification datasets with 50 or fewer features, of which eight or fewer numeric features and no large text features, with no missing values, with at most

[1]All code is part of the online appendix of this thesis: `https://doi.org/10.4121/3584f9cd-cf7b-478c-960f-3b2ef47d1427`.

100,000 instances, and at least 40 instances. We take the most recent version of the dataset and omit duplicates or datasets that only differ in the random seed, resulting in 180 datasets. We split each dataset into five folds, creating five train and test pairs each consisting of four and one fold respectively. We list all datasets used in Appendix 7.E and also include a histogram of the dataset sizes.

We implemented all the ODT objectives in the ODT method STreeD (Van der Linden et al., 2023) because of its scalability and flexibility in supporting new objectives.[2] STreeD is a DP approach that requires binary features. Therefore, we binarize the numeric training data with thresholds on the ten quantiles, and the categorical data with one-hot encoding (with at most ten categories). The test data is binarized in the same way. We experimented with other binarization approaches, but noted no significant impact with regard to the analysis presented here. See Section 7.4.2 for the impact of binarization on CART. See Appendix 7.D.2 for a brief evaluation of the impact of binarization on ODTs.

Additionally, we evaluate on synthetic datasets where we can control the amount of noise. We follow the synthetic data setup from Murthy and Salzberg (1995) and Bertsimas and Dunn (2017) and Dunn (2018). We generate n random training instances with p numeric features, uniformly distributed over $[0, 1]$. For a given noise strength f, we add feature noise by adding noise uniformly drawn from $[-f, f]^p$. Again, we binarize the numeric features by threshold predicates on 10 quantiles per numeric feature. We generate a random binary tree on this binarized data of a maximum depth d with at most 2^d leaf nodes. We choose random splits on the data such that each leaf node contains at least 5 instances. The binary labels of each leaf node are assigned alternately, such that no split leads to two leaf nodes with the same label. After this, we add class noise to a given percentage c of the data by flipping its label. For each training set, we create a corresponding test set without noise of 1000 instances per leaf node in the generated tree.

While keeping the other values constant, we test with changing the amount of feature noise ($f = 0$, 0.2, 0.4, 0.6, 0.8, 1.0) and the amount of class noise ($c = 0\%$, 10%, 20%, 30%, 40%, 50%), while also changing the number of instances ($n = 50$, 100, 1000). We repeat each configuration 1000 times and report averages over these 1000 runs.

We train ODTs up to depth four while tuning the number of branching nodes using cross-validation.[3] In cross-validation, for datasets with up to 100 instances, we use 20 folds; for up to 250 instances, we use ten folds; and otherwise, we use five.

We test the objectives on TDI heuristics in our own implementation of CART. We use cost-complexity tuning with accuracy as the pruning objective. For comparison with ODT, we train CART using a depth limit of four on the same binarized datasets. In Section 7.4.2, we evaluate the impact of these choices.

The average rank is our main performance metric: for each dataset split, we round the test accuracy to one decimal and then rank all methods. If multiple methods have the same accuracy, they are all assigned the average rank. E.g., if two

[2] `https://github.com/algtudelft/pystreed`

[3] A depth-three tree has at most eight leaf nodes and seven branching nodes and a depth-four tree has at most 16 leaf nodes and 15 branching nodes.

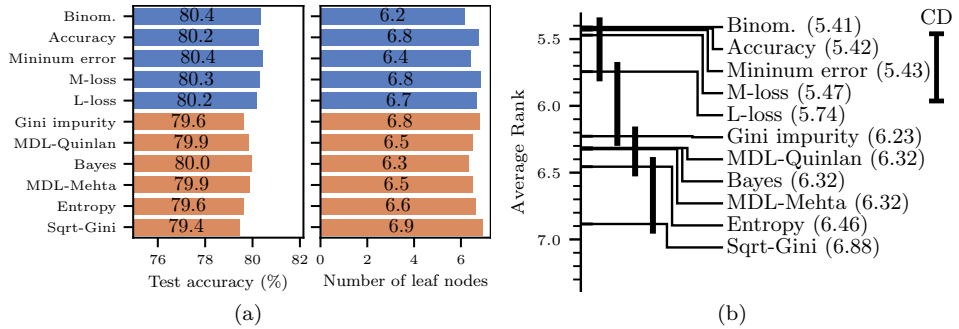


Figure 7.4: Comparing ODT objectives for max-depth = 4. (a) Orange (blue) indicates (non-)concave. The average accuracy and number of leaf nodes over all datasets and folds are shown, sorted by the average rank. (b) Nemenyi critical distance rank test. The average rank per objective is plotted and objectives with a rank difference smaller than the critical distance (CD) at p-value 0.05 are grouped by a black bar.

methods have the same best score, they both get rank 1.5. We then report the mean rank over all datasets.

Optimal Decision Tree Results

Fig. 7.4 shows the average rank, test accuracy, and the number of leaf nodes per objective for trees of depth four on the OpenML benchmark. On average, the best ranking objectives with the best test accuracy are accuracy and its slight variations minimum error and the binomial pessimistic error, and the novel non-concave functions M-loss and L-loss. Maximizing training accuracy is ranked second, although the difference from the top objective is not significant. The pessimistic binomial objective achieves the best test accuracy with a lower average number of leaf nodes than the accuracy objective.

However, the results are close. Fig. 7.4b shows the results of a Nemenyi critical distance rank test to test the significance (Demšar, 2006). This test computes the critical distance (CD) between the average ranks of two methods to be statistically significant. Fig. 7.4b shows that all the non-concave objectives are not significantly different for depth four. All the non-concave objectives are significantly better than all the concave objectives.

The runtimes of optimizing the objectives are close: the objective with the lowest runtime (L-loss) is on average 1.9 times faster (geometric mean) than the slowest objective (Bayes).

In Appendix 7.D.1, we show training accuracy results for a selection of datasets for three objectives (accuracy, Gini impurity, and M-loss) for an increasing number of nodes.

Greedy Heuristics Results

For comparison, we also train greedy trees with the objectives listed above. Fig. 7.5 shows the resulting out-of-sample performance. In comparison with the ODTs,

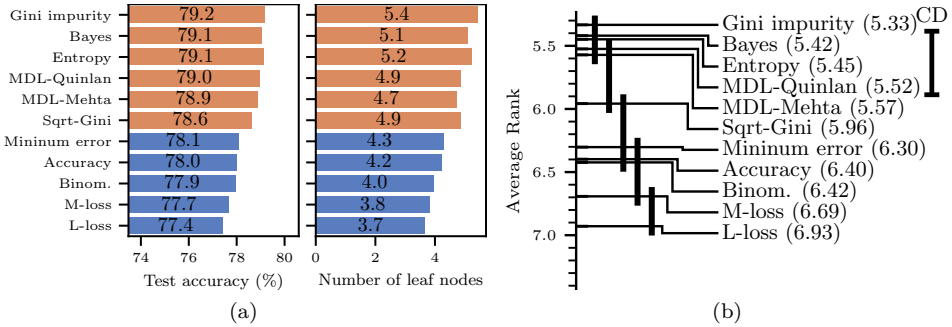


Figure 7.5: Comparing greedy objectives for max-depth = 4. The strictly concave objectives (orange) significantly outperform the non-concave objectives (blue).

the ranking of the objectives is almost reversed, which is in line with our analysis in Section 7.2.2, that the TDI approach requires strictly concave objectives. The Nemenyi critical distance rank test in Fig. 7.5b shows that all strictly concave objectives (except square-root Gini) are significantly better than all non-concave objectives for TDI heuristics. The smaller tree sizes of the non-concave objectives show that the greedy heuristic gets stuck early with these objectives and finds no improving splits. Optimizing Gini impurity yields 1.2% higher out-of-sample accuracy than directly optimizing the in-sample accuracy. These results confirm that the traditional Gini and entropy are among the top choices.

Noisy Synthetic Data

To further investigate the performance of the ODT objectives, we compare their relative performance on synthetic data for varying dataset sizes and levels of feature and class noise. We limit the comparison to the top four objectives and include Gini impurity as the top representative of the concave objectives.

Fig. 7.6a shows that for the smallest datasets ($n = 50$) with an increasing level of feature noise, the binomial pessimistic and the minimum error obtain (statistically) significantly better test accuracy (+1%) than the other objectives. Gini impurity performs slightly worse. With more data ($n = 100, 1000$), the advantage of the binomial pessimistic and minimum error disappears, whereas Gini impurity performs relatively even worse, up to 2%.

Fig. 7.6b shows that the differences for increasing amounts of class noise are smaller. However, Gini impurity still performs worse than the other objectives, especially for larger datasets ($n = 1000$) and more class noise. When the class noise reaches 50%, half of the labels are flipped and the training labels essentially are completely random. That is why the relative test accuracy for 50% class noise goes back to zero.

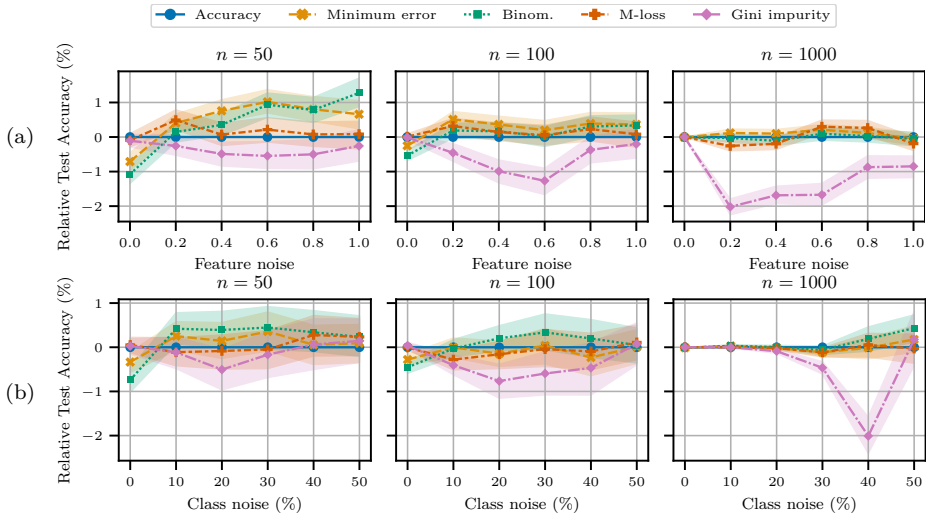


Figure 7.6: Relative test accuracy of ODT objectives compared to optimizing training accuracy directly. The concave Gini impurity performs significantly worse. For small datasets with feature noise, the binomial pessimistic and minimum error perform significantly better (the error bar represents the 95% confidence interval).

7.2.5. Discussion

Previous work has extensively compared greedy splitting criteria, which we extend to optimal decision trees. Our results show that optimizing accuracy *directly* is a good choice for ODTs, specifically when the number of training samples increases because, with sufficient training data, the training accuracy closely approximates the test accuracy. This shows that the strict concavity of objectives such as Gini impurity and entropy, is not an inherently necessary or desirable property, but a limitation imposed by the greedy TDI approach. Optimal and greedy training procedures respond differently to the objectives and therefore best practices of one approach do not necessarily translate to the other. This opens the question of how the performance of non-strictly-concave objectives for small datasets could be exploited in greedy heuristics.

We hypothesized that non-concave objectives may perform better than accuracy for noisy data. Although we did not measure a significant difference for the new objectives M-loss and L-loss, we found two objectives that can outperform optimizing accuracy directly for small noisy datasets: the pessimistic binomial and minimum error objective. Both of these objectives include a regularizing component, which –in addition to the regularization effect of tuning the tree size– helps to prevent overfitting. This also results in slightly smaller models, while retaining the same average accuracy.

7.3. Tuning the Complexity of ODTs

Most decision tree learning approaches make a trade-off between training accuracy and model complexity to prevent overfitting. For ODTs, several complexity tuning methods have been used but without an in-depth empirical comparison. Therefore, this section analyses the effect of complexity tuning methods for optimal decision trees, starting with an overview of complexity tuning methods. We find that optimal decision trees perform significantly better with tuning than without, but that existing tuning techniques perform similarly.

7.3.1. Tuning Approaches

Currently, ODT approaches mostly tune the following hyperparameters: tree depth (Aglin et al., 2020a), tree size (Demirović et al., 2022), complexity cost (Lin et al., 2020), and the minimum support (Nijssen and Fromont, 2007). In Appendix 7.B, we provide additional information on these approaches. On top of the existing ODT tuning approaches, we evaluate two new approaches:

Question length: The *question length* counts the average number of tree nodes visited by an instance to be classified and is shown by Piltaver et al. (2016) to be one of the best proxies for human comprehensibility of trees. Question-length cost tuning minimizes $\omega \sum_{b \in B} |b| + \sum_{(n,e) \in L} f(n,e)$, with L the set of leaves, B the set of branching nodes, $|b|$ the number of instances passing through a branching node b, and ω the question-length cost parameter.

Smoothing: Because of the good performance of the minimum error objective in Section 7.2.1, we generalize this approach by tuning the Laplace smoothing parameter. The Laplace smoothing approach (Flach, 2012) assumes in a leaf node that for each class, x extra instances exist. With $|\mathcal{K}|$ the number of classes, the accuracy objective becomes

$$f(n,e) = \frac{n(e+x)}{n+|\mathcal{K}|x}\,.$$

7.3.2. Experiments

The experiments address the following questions: how do the ODT tuning methods compare in out-of-sample accuracy and how do they respond to increasing noise?

Experiment setup

We evaluate each tuning method on the same 180 OpenML and synthetic datasets of Section 7.2.4. Regardless of the tuning method, we impose a maximum depth constraint of four or five. For each tuning method, we select at most k different parameter settings. For depth, these settings are selected using equal linear spacing between the k options. For the others, the k settings are selected using equal spacing in the log scale. In the results presented here, we set $k = 16$ (because a depth-four tree can have zero to fifteen branching nodes, i.e., 16 options). We provide more details

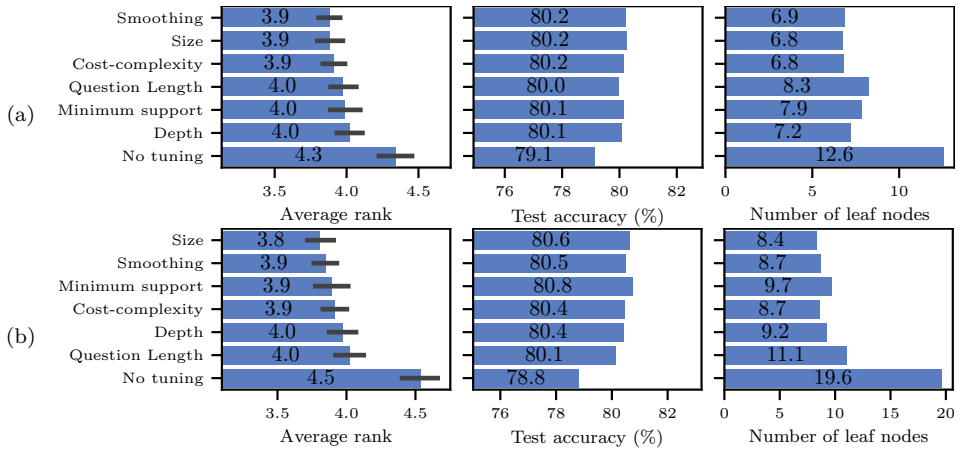


Figure 7.7: Complexity tuning results for ODTs of (a) max-depth = 4 and (b) max-depth = 5 for five runs on (a) 180 datasets and (b) 157 datasets.

on how the values are chosen and experiments for other values of k in Appendix 7.B. All tuning methods are implemented in STreeD (Van der Linden et al., 2023).

For most datasets, we obtain trees well within the maximum depth limit of five, and therefore we do not extend this analysis to larger depth limits.

Results on Real Datasets

Figs. 7.7a and 7.7b show the performance of the complexity tuning method on the OpenML datasets. We exclude datasets from the analysis if any method exceeded the two-hour time-out: for depth four, no datasets are removed, and for depth five, 23 datasets. Surprisingly, the results show that all tuning methods obtain similar accuracies: there is no statistically significant difference between any of the approaches. In terms of optimizing accuracy, the only conclusion is that using any of the tuning approaches is better than no tuning.

However, other differences can be observed between the methods. For example, tuning the question length, minimum support, or depth yields slightly larger trees. Furthermore, in Appendix 7.B, we measure the runtime and test the performance of the methods for different numbers of parameter settings k. This shows that tuning only the depth of the tree is by far the fastest approach.

Inspecting the results shows that the differences among tuning methods are largest for medium-sized datasets (several hundred to ten thousand samples). For example, on the 82 datasets with 250 or more and 10,000 or fewer samples, a Wilcoxon signed rank test shows tuning the size is statistically significantly better than tuning the depth (p-value ≈ 0.01).

Results on Synthetic Data

To further test the tuning methods' performance, we evaluate them on the synthetic datasets. Fig. 7.8a shows that as the amount of feature noise increases, tuning

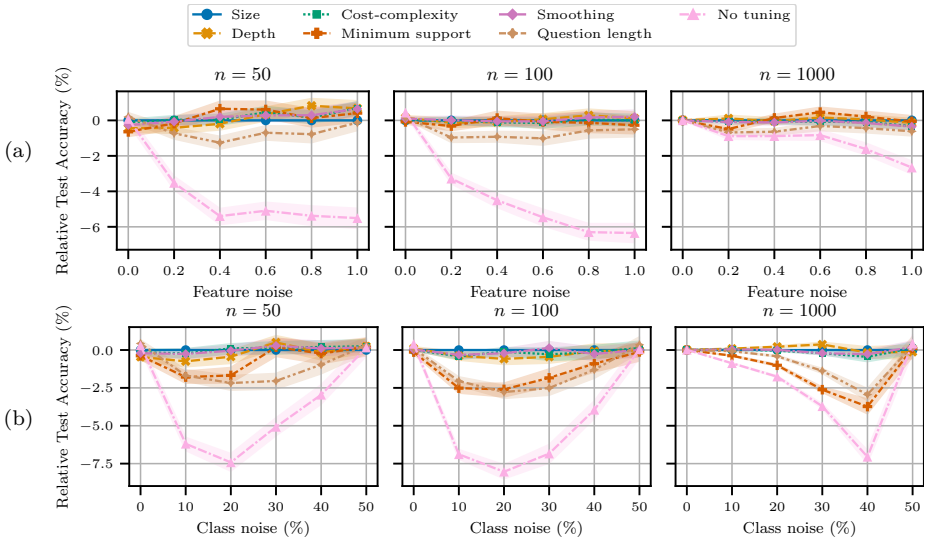


Figure 7.8: Relative test accuracy of ODT tuning methods compared to tuning the size (number of nodes) of the tree. No tuning, tuning the minimum support, and tuning the question length are significantly worse than the other tuning approaches when noise is present.

becomes increasingly more important. It also shows that tuning the question length is slightly worse than the other tuning methods. No clear differences can be observed between the other tuning methods.

Fig. 7.8b shows larger differences when increasing the amount of class noise. Again, not tuning is significantly worse when noise is present. Tuning the minimum support and the question length is significantly worse than the other tuning methods when class noise is present. At 50% class noise, this difference obviously disappears because then all training labels are basically decided randomly.

Interestingly, Fig. 7.8 shows no significant difference in tuning only the depth versus the more expensive procedures of tuning the size, the cost-complexity, or the smoothing level.

7.3.3. Conclusion

In conclusion, complexity tuning of ODTs is necessary. On our real datasets, all previously used tuning approaches, obtain similar accuracy results. On the noisy synthetic data, on the other hand, tuning the question length or minimum support is significantly worse than the alternatives. Tuning the depth is more time efficient than other approaches, obtains the same accuracy, but does so with slightly more nodes. One new tuning approach is promising: tuning the amount of smoothing. Based on our experiments in Sections 7.2 and 7.3, we recommend training ODTs as follows:

Recommendations 1 (Training Optimal Decision Trees)**.**

1. Optimize the same loss at train and test time for optimal decision trees.
   Optimizing the accuracy on average yields the best out-of-sample accuracy. Do not optimize concave proxies such as Gini impurity or entropy.

2. In noisy datasets, consider an objective with an additional regularizing effect.
   In noisy datasets, optimizing objectives such as the pessimistic binomial and minimum error may perform better than maximizing accuracy as they have an additional regularization effect by encouraging model sparsity.

3. Tune the complexity of optimal decision trees.
   Training ODTs with hyperparameter tuning is significantly better than training without hyperparameter tuning.

4. Tune the size, complexity cost, or smoothing parameter; or, in the case of large datasets, the depth.
   Tuning the size, complexity cost, smoothing, or depth, on average, yields similar out-of-sample results. However, tuning the depth yields larger trees. For large datasets, tuning is less important, and tuning the depth is more runtime efficient.

7.4. Comparing Optimal and Greedy Decision Trees

To understand the differences between greedy and optimal approaches for learning decision trees, we collected claims from the literature and evaluated these claims with extensive experiments. Below, we list the claims discussed in this section and summarize our results:

Claim 1: Optimal methods under the same depth constraint (up to depth four) find trees with 1-2% higher out-of-sample accuracy than greedy methods (Bertsimas and Dunn, 2017; Verwer and Zhang, 2017; Demirović et al., 2022).

Supported: We evaluate the accuracy of depth three and four trees on 180 datasets and find an average improvement of 1.3% and 1.0% of optimal over greedy approaches.

Claim 2: Optimal methods obtain a better accuracy-interpretability trade-off than greedy methods (Lin et al., 2020).

Supported: We evaluate the accuracy of trees with 1 to 16 nodes on 180 datasets and find that the size-weighted accuracy of optimal methods is, on average, significantly higher than that of greedy methods. The size-weighted accuracy is a new metric we propose in Section 7.4.3 to measure the accuracy-interpretability trade-off.

Claim 3: The difference between optimal and greedy approaches diminishes with

more data (Murthy and Salzberg, 1995; Costa and Pedreira, 2023).

Refuted: Experiments on synthetic and real data show that size-constrained greedy trees do not improve after some point and stay worse than optimal. In contrast, the performance of size-unconstrained greedy trees keeps improving with more samples, while also growing much larger trees than ODTs.

Claim 4: The accuracy of greedy trees remains stable when the data size increases linearly with concept complexity (Murthy and Salzberg, 1995).

Refuted: On synthetic data generated from a random decision tree, the performance of optimal decision trees remains stable when the random tree's size is increased, and the performance of greedy trees diminishes.

Claim 5: Optimal trees are more likely to overfit than greedy trees (Dietterich, 1995).

Refuted: With hyperparameter tuning, we do not find significant performance differences between optimal and greedy methods with small numbers of samples (up to 250) nor more sensitivity to noise.

Claim 6: The *question length* of greedy trees remains (in practice) close to the optimal tree depth (Goodman and Smyth, 1988; Murthy and Salzberg, 1995).

Supported: Our experiments on synthetic data show that the question length of optimal and greedy methods remains similar in (almost) all (practical) scenarios. Only for very noisy data, does CART yield much longer question lengths.

In this section, we first review previous greedy-optimal comparisons, which we use to design a set of best practices for future comparisons. The rest of the section details the experiments we performed to evaluate each of the claims from existing literature. In Section 7.4.5 we discuss the scalability of ODTs.

7.4.1. Previous Comparisons

In Appendix 7.C, we list previous comparisons between greedy and optimal approaches. These comparisons can be grouped into ODT papers that propose new ODT methods, and other papers. From these comparisons, the following general observations can be made about how greedy and optimal methods are compared:

- Both greedy and optimal methods are often not correctly tuned, or not tuned at all. When comparing with CART, many papers show several modifications to how CART is trained and tuned. We assess the impact of each of these modifications on the accuracy in Section 7.4.2 below. Additionally, several papers evaluated ODTs without tuning the complexity. As shown in Section 7.3.2, this significantly worsens the ODT performance. Others evaluate ODTs with a substantial restriction on the tree size, resulting in shallow underfitting trees.
- Almost all ODT papers compare with CART under the same tree-size constraint and draw a positive conclusion on the ODTs' performance. The other papers

Year	Author(s)	Method	Greedy with same size limit	Greedy without size limit	Small and large datasets	Beyond small trees	Optimal tuned (correctly)	Greedy tuned (correctly)
Papers that propose ODT methods								
2007	Nijssen and Fromont	DL8	✓	✓		✓		
2017	Bertsimas and Dunn	OCT	✓	✓		✓	✓	✓
2019	Verwer and Zhang	BinOCT	✓			✓		
2020	Lin et al.	GOSDT				✓	✓	
	Hu et al.	MaxSAT	✓			✓		
2021	Günlük et al.	ODT	✓				✓	✓
2022	Demirović et al.	MurTree	✓		✓	✓		
	Hua et al.	RS-OCT	✓		✓	✓	✓	✓
	Mazumder et al.	Quant-BnB	✓		✓			
2024	Liu et al.	BNP-OCT	✓		✓			
	Alès et al.	CTT		✓		✓	✓	✓
2025	Brița et al.	ConTree	✓		✓		✓	✓
Other papers that compare ODTs with greedy DTs								
1995	Murthy and Salzberg	-				✓		✓
2021	Zharmagambetov et al.	-		✓	✓	✓		✓
2024	Sullivan et al.	MAPTree		✓		✓		
	Marton et al.	GradTree		✓	✓	✓		

Table 7.2: Simplified overview of the comparisons between greedy and optimal methods in the literature. Ideally, a comparison checks all columns. 1) Compare methods under the same size constraint; 2) compare (greedy) methods without a size constraint; 3) compare on small and large datasets ($>$ 10.000 instances); 4) compare optimal methods beyond depth three; 5) tune optimal methods (correctly); and 6) tune greedy methods (correctly).

typically compare ODTs under a size constraint with unconstrained non-optimal approaches and sometimes draw negative conclusions about the ODTs' performance. Both comparisons are useful for different purposes. One simply evaluates out-of-sample performance, while the other evaluates the claim that ODTs have a better accuracy-interpretability trade-off. We further discuss measuring this trade-off in Section 7.4.3 below.

- Several comparisons are limited to small datasets and small trees (e.g., less than 10000 instances or trees of maximum depth three). This is typically because of scalability limitations. The improved scalability of optimal methods allows us to analyze larger datasets and trees.

Table 7.2 summarizes our observations about previous comparisons. We recommend that future comparisons between ODTs and greedy approaches should check all the columns in this table. We summarize our recommendations as follows:

Recommendations 2 (Greedy-Optimal Comparisons)**.**

1. Compare both with and without constraining the sizes of the decision trees.
   Optimal decision trees optimize performance for a particular size; therefore, greedy methods should be equally constrained in size to compare fairly.
2. Compare performance on both small and large datasets.
   While experiments on small data are more efficient to run, their results often do not carry over to larger datasets.
3. Evaluate both small and large trees.
   Previous comparisons often compare trees of depth two. This optimization problem is too simplistic, and the results do not always carry to a larger depth.
4. Tune both greedy and optimal methods and ensure an equal comparison.
   Comparisons between greedy and optimal trees at a fixed depth are unequal since the methods respond differently to size constraints. If one wants to compare trees up to depth four, for example, both approaches should be tuned up to a maximum depth of four, not trained with a fixed depth of four.

7.4.2. Training CART

In the comparisons reviewed in Appendix 7.C and summarized above, ODTs are often compared to a modified version of CART, for example, to allow for a direct comparison under similar circumstances. We test the impact of these modifications to assess the validity of these previous comparisons and inform future comparisons. We assess the following typical modifications: 1) tuning the depth instead of the complexity cost; 2) binarizing the feature data; or 3) running CART while imposing an additional depth constraint.

We compare CART's performance with these modifications against unmodified CART on the 180 OpenML datasets used before. We approximate the unconstrained CART with a maximum depth of 20. We set the constrained depth limit to four, because of its common use in ODT comparisons. As before, the binarized data has up to ten binary features per continuous feature by using thresholds on ten quantiles or one-hot encoding of categorical features with a maximum of ten categories.

In addition to the depth limit, we apply cost-complexity pruning as done in RPart (Therneau et al., 2023): we train a fully expanded tree and obtain the cost-complexity path with all possible cost-complexity values from that tree and use the geometric mean to get the midpoints of those values. We use cross-validation to find the best cost-complexity parameter among the midpoints and retrain a tree on the full training data with this parameter.

Unlike RPart, we take the best performing cost-complexity parameter, and not

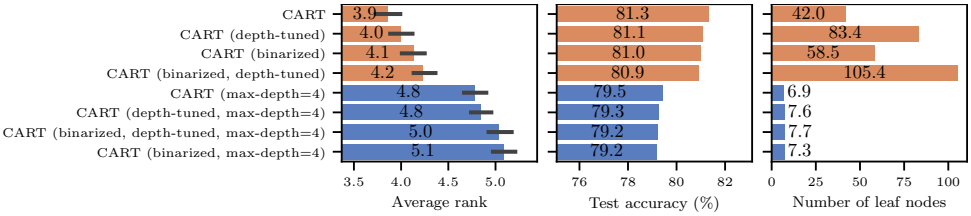


Figure 7.9: Results for CART (Gini) when trained with(out) binarization, with(out) a depth limit, and when using depth or cost-complexity tuning. Using a depth limit (as indicated in blue) significantly impacts performance. Tuning the depth has a more negative impact for large maximum depths. Binarization with 10 quantile thresholds has no significant impact on the accuracy but does impact the tree size.

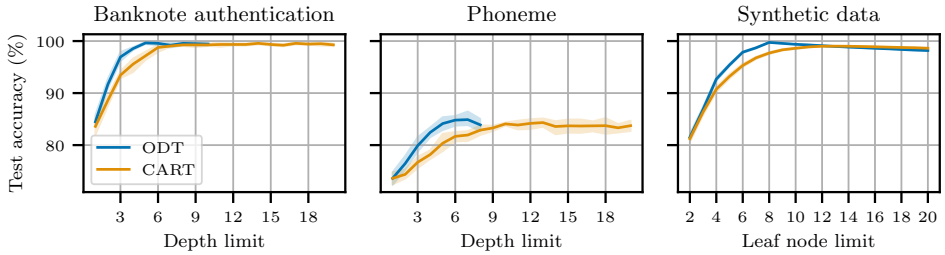


Figure 7.10: Typical accuracy-interpretability trade-off for untuned greedy and optimal decision trees. ODTs have a slight advantage for small size limits but both methods converge for large size limits.

the largest complexity cost which performs within one standard error of the best performing one. In our preliminary tests, this resulted in larger trees but better out-of-sample accuracy.

Fig. 7.9 shows CART's performance under these modifications. The largest differences are between the depth-constrained and the unlimited depth variant. Binarization has only a small impact on the performance (this does not necessarily generalize for more coarse binarizations). When a strict depth limit is imposed, tuning the depth instead of the complexity cost has a small impact but for the unlimited depth case, this significantly hurts CART's performance. Fig. 7.9 also shows significantly different tree sizes for CART's modifications. Both binarization and depth tuning result in larger trees. From these results, we can conclude the following best practices:

Recommendations 3 (Training CART)**.**

1. Training CART with a depth limit should be clearly stated.
   CART trained with a depth limit results in significantly different results than CART without a depth limit.
2. Tuning the depth of CART instead of the complexity cost should be avoided.
   Tuning CART's depth rather than the cost-complexity yields larger trees.
3. Training CART on binarized data should be clearly stated.
   Depending on the binarization, training on binarized data may or may not significantly harm the performance.

7.4.3. Accuracy-Interpretability Trade-Off

ODT papers typically compare ODTs with greedy heuristics under a similar size constraint. This is partly motivated by the definition of ODTs because ODTs are defined as trees that maximize *training* performance *under a given size limit*. No theoretical claim is made about its out-of-sample performance or the performance without a size constraint. In fact, finding an optimal tree without a size constraint is trivial: it is obtained by splitting in any way until no further split can be made.

It is also motivated by the claim that ODTs have a better accuracy-interpretability trade-off. Since one of the oft-cited motivations for decision trees is their comprehensibility and large trees with hundreds of nodes can hardly be called human-comprehensible (Piltaver et al., 2016), evaluating an algorithm's ability to obtain small performant trees is important.

Typically, the tree size and test accuracy are plotted against each other, as done in Fig. 7.10, to assess this trade-off. Such plots show the relative performance for different size limits and also the saturation point: when adding more nodes gives no improvement in accuracy. Optimal methods typically reach this saturation point earlier and greedy methods eventually catch up by obtaining the same accuracy but with larger trees.

Such comparisons require one figure per dataset. To enable easier comparisons across a large number of datasets, we propose a new metric: the size-weighted accuracy (SWA). The purpose of this metric is to express the accuracy-interpretability trade-off with one number that represents the 'surface' under the accuracy-interpretability Pareto front seen in Fig. 7.10. Since we are mostly interested in the performance of *small* trees, we set the weight of the accuracies obtained for a tree with i leaf nodes to $1/i$. We define the size-weighted accuracy (SWA_n) as the weighted average of the obtained trees of maximum size n:

$$\text{SWA}_n = \frac{1}{\sum_{i=1}^{n} \frac{1}{i}} \sum_{i=1}^{n} \frac{\text{acc}_i}{i} \,. \tag{7.1}$$

Since not every algorithm can directly set the number of leaf nodes, we propose to

run the algorithm with different complexity parameters (e.g., the complexity-cost parameter λ) and record the resulting number of leaf nodes and test accuracy for each run. If multiple runs yield the same number of leaf nodes, then average these test accuracies. For missing tree sizes, linearly interpolate test accuracies using results from the nearest smaller and larger tree sizes. If the largest tree size obtained is less than n, assign the test accuracy of this largest tree to all larger sizes up to n. If larger trees result in smaller test accuracy, replace it with the larger value for smaller trees, since we measure the Pareto front.

For example, for the synthetic data in Fig. 7.10, we computed trees up to 20 leaf nodes. CART obtains a SWA_{20} of 84.0%, whereas the optimal approach obtains a SWA_{20} of 84.7%.

7.4.4. Experiments on Literature Claims

To evaluate the claims made in previous papers, we compare optimal and greedy methods on synthetic and real datasets, using the best practices introduced in Recommendations 1-3.

Experiment Setup

We again evaluate on both the OpenML and synthetic datasets introduced in Section 7.2.4. We add two more real datasets to evaluate Claim 3: *covertype* and *Higgs*. These two were chosen for their large number of samples, allowing us to investigate performance under various sample sizes (566,602 and 940,160 respectively). For the synthetic data, unless otherwise specified, we set the true tree depth to three, the number of instances $n = 1000$, the number of numeric features $p = 3$, the feature noise $f = 0$, and the class noise $c = 0\%$. We test with changing the number of instances ($n = 50$, 100, 250, 1000, 10000), the number of features ($p = 2$, 4, 6, 8), and with changing the noise as done in Section 7.2.4. Additionally, we add synthetic datasets with a linear separator instead of a tree as the ground truth. The weights of the linear separator are chosen randomly from a normal distribution. We repeat each configuration 500 times and report averages over these 500 runs.

We evaluate CART and ODT on the binarized data to eliminate this difference between the two methods and focus only on the difference between greedy and optimal search. Comparing on binarized data is commonly done (Lin et al., 2020; Demirović et al., 2022). Section 7.4.2 shows that the impact of this binarization on CART is small for the datasets in our benchmark. In Appendix 7.D.2, we also evaluate the effect of binarization for ODTs.

In our synthetic tree experiments, apart from the familiar test accuracy and number of leaf nodes, we also measure the following:

True Discovery Rate (TDR): The TDR is the percentage of splits in the ground truth tree that are recovered in the trained tree (higher is better).

False Discovery Rate (FDR): The FDR is the percentage of the splits in the trained tree that are not part of the ground truth tree (lower is better).

Question Length: The question length is the average number of branching nodes an instance visits when evaluated (lower is better).

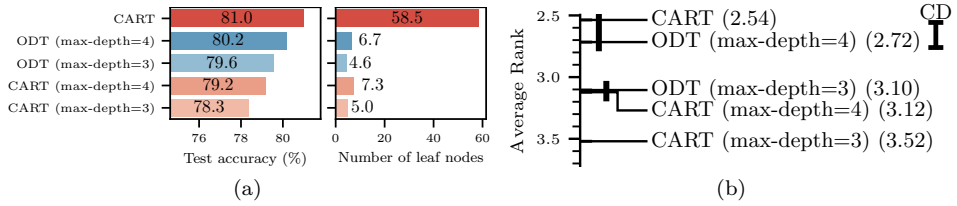


Figure 7.11: Out-of-sample accuracy of CART and ODTs compared on five runs for 180 datasets. (a) Optimal (blue) versus CART (red). CART without a depth limit performs best but yields much larger trees. (b) Nemenyi critical distance rank test for optimal versus CART. The average rank per method is plotted and methods with a rank difference smaller than the critical distance (CD) at p-value 0.05 are grouped by a black bar. With the same depth limit, optimal performs significantly better than CART, confirming Claim 1.

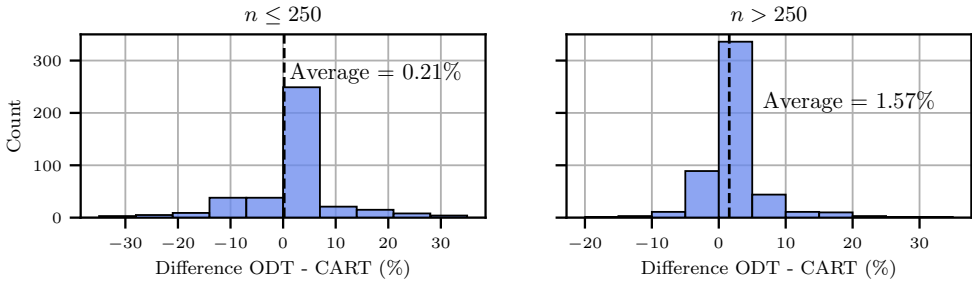


Figure 7.12: The difference between the optimal and CART out-of-sample accuracy for both small (left) and larger (right) datasets. The dashed line indicates the average difference.

7

Out-of-Sample Accuracy

Claim. Optimal methods under the same depth constraint (up to depth four) find trees with 1-2% higher out-of-sample accuracy than greedy methods (Bertsimas and Dunn, 2017; Verwer and Zhang, 2017; Demirović et al., 2022).

To evaluate Claim 1, we evaluate both the ODT approach and CART on the OpenML datasets with a depth limit of three and four. We also compare with CART without a depth limit. Fig. 7.11 shows that CART (without a depth limit) performs better (but not significantly) than the ODT approach with a depth-four depth limit, but CART yields much larger trees.

When compared under the same depth constraint, as stated in Claim 1, optimal significantly outperforms greedy with an average improvement of 1.3% and 1.0% for depths three and four respectively. Since optimal algorithms optimize the complete decision tree, instead of greedily improving the tree, they can achieve better scores.

Fig. 7.12 shows the distribution of the differences between optimal and greedy when trained with maximum depth four. For small datasets ($n \leq 250$), the average advantage of the optimal approach is $0.2\% \pm 0.4$ (mean $\pm$ standard error). Some large accuracy differences occur because the test sets for these small datasets are so small that a single misclassified instance can increase or decrease accuracy by several percent. For larger datasets ($n > 250$), the average improvement of optimal

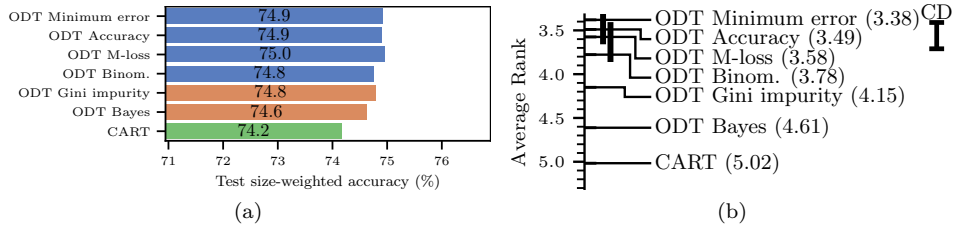


Figure 7.13: Comparing the test size-weighted average (SWA) of several ODT objectives and CART (see Section 7.4.3). (a) The non-concave objectives are blue, the concave objectives are orange, and CART is green. (b) The ODT minimum error, accuracy, and M-loss are significantly better than the other approaches. All ODT approaches are significantly better than CART, confirming Claim 2.

over CART is $1.6\% \pm 0.2$. The difference is larger and the standard error is lower. These results confirm Claim 1.

Accuracy-interpretability trade-off

Claim. Optimal methods obtain a better accuracy-interpretability trade-off than greedy methods (Lin et al., 2020).

To test Claim 2, we compute the size-weighted accuracy (SWA) introduced in Section 7.4.3 to measure the surface under the accuracy-interpretability Pareto front. We train ODTs with the top four non-concave and top two concave objectives from Section 7.2 and CART with the traditional Gini impurity objective. ODTs are trained with a maximum depth of four and CART is trained without a depth limit.

Fig. 7.13 shows that on average the optimal approach achieves a higher SWA than CART, thus verifying Claim 2. For large datasets, the training accuracy is close to the test accuracy for highly regularized models which means that optimal decision trees reliably improve over greedy trees. This result also repeats the conclusion of Section 7.2.4, that non-strictly-concave objectives are to be preferred over concave objectives for optimal methods.

Data Efficiency

Claim. The difference between optimal and greedy approaches diminishes with more data (Murthy and Salzberg, 1995; Costa and Pedreira, 2023).

To test Claim 3, we evaluate ODT and greedy performance on large real datasets and synthetic data with an increasing number of training samples and features.

Fig. 7.14a shows how ODTs compare with CART for an increasing number of training instances on synthetic data generated from ground-truth trees of depth three. For less than 1000 instances, the ODTs are more accurate than both CART with and without a maximum depth limit. For more than 1000 instances, both ODT and CART obtain 100% test accuracy. CART, however, uses 11 leaf nodes to achieve this result, whereas the optimal approach only requires eight (equal to the true tree's complexity). Both approaches have approximately the same true discovery rate.

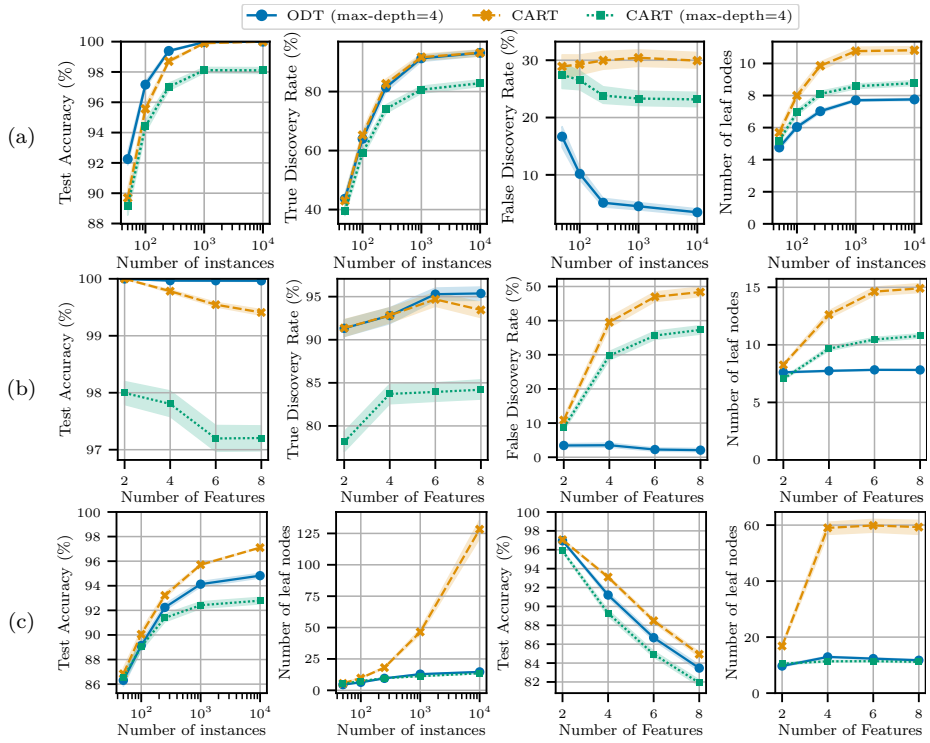


Figure 7.14: Results on the synthetic tree datasets for increasing (a) number of training samples, and (b) number of numeric features; and (c) on the synthetic linear datasets for increasing number of training samples and numeric features, to evaluate Claim 3.

However, ODT's false discovery rate is lower. More instances help the ODT method to reduce its false discovery rate.

The depth-constrained CART's test accuracy plateaus around 1000 training instances at 98%. This shows that CART requires a higher depth limit to obtain the same accuracy as ODTs, regardless of how much data it receives. Even though the true tree depth is three, a maximum depth of four for CART is not enough to recover the tree.

Fig. 7.14b shows how CART's accuracy drops when the number of features in the synthetic data increases whereas the optimal approach retains the same perfect accuracy. This difference can be explained by observing the rise in the FDR of CART when the number of features increases together with an increase in the number of leaf nodes: it finds more unnecessary splits. This shows that CART performs worse for an increasing number of features, whereas the optimal approach remains unaffected.

Fig. 7.14c additionally shows that when learning from synthetic data with a linear separator as the ground truth, CART without a depth constraint achieves higher accuracy than ODT, but with more data availability, CART also generates much larger trees with only a small gain in accuracy. Increasing the number of features in

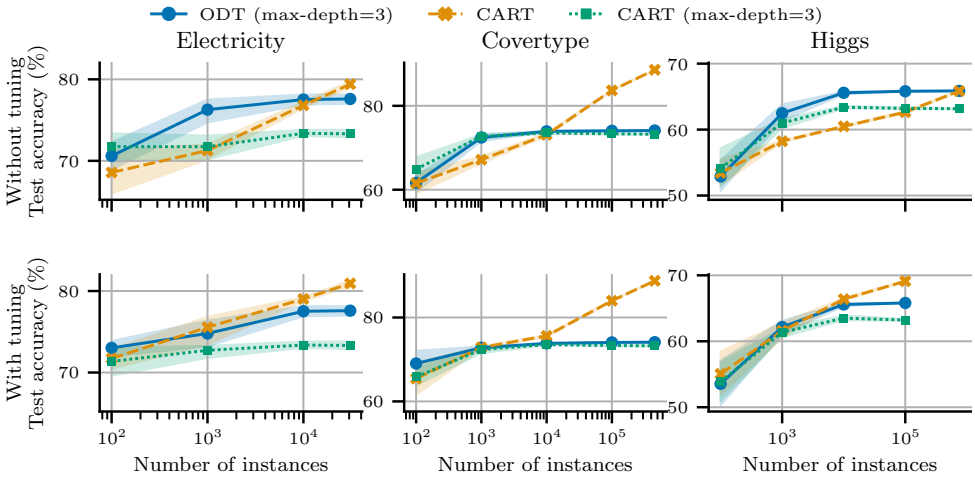


Figure 7.15: Test accuracies for increasing training samples with and without tuning.

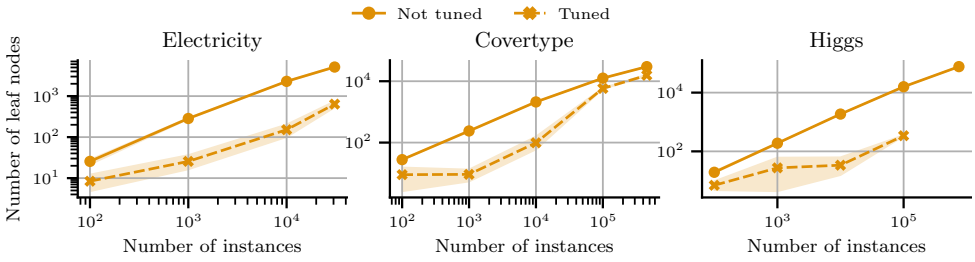


Figure 7.16: Number of leaf nodes for CART for increasing training samples with and without tuning. These numbers far exceed the eight leaf nodes for ODTs in Fig. 7.15.

the synthetic data makes the classification function harder to learn. The relative accuracy performance of the methods stays roughly the same, but CART requires many more nodes. In all cases, ODTs perform better than CART with the same depth constraint.

Fig. 7.15 shows the out-of-sample accuracies for increasing training sample sizes on three large real datasets. We train ODTs with a depth limit of three, and CART with and without a depth limit of three. When the methods are *not* tuned, the optimal approach overfits on small datasets, obtaining a lower test accuracy than depth-limited CART. However, *with* tuning, this effect disappears and the ODTs' accuracy is consistently higher than CART's (depth limited). The difference in performance between tuning and not tuning diminishes for larger training sets. Without a depth limit, CART continues to increase its accuracy for more data. More data does not help depth-limited CART since, at some point, the greedy decisions on what feature and threshold to split on do not change anymore. In those cases, the added data does not lead to different greedy decisions but only makes them more certain.

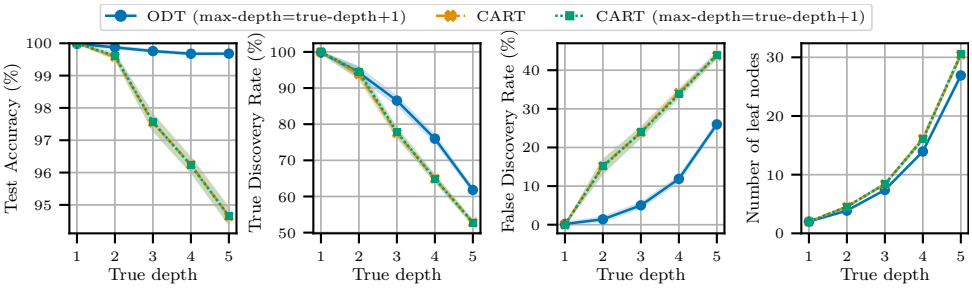


Figure 7.17: Testing Claim 4 on the synthetic datasets for ground truth trees of increasing complexity and training samples $n = 50 \cdot 2^d$, with d the depth of the ground truth tree.

However, Fig. 7.16 also shows that CART continues to grow larger trees with up to tens of thousands of nodes. For the Electricity dataset, for example, CART and ODTs have roughly similar accuracy for ten thousand training samples. However, the ODT has eight leaf nodes, whereas CART has over a hundred. These results contradict the observation by Oates and Jensen (1997) that greedy tree methods do not perform much better for more data but do yield larger trees with more data. For these datasets, we observe CART performing much better for more data while also resulting in larger trees.

In conclusion, these results refute Claim 3 that the difference between optimal methods and greedy diminished for more data. CART (without a depth limit) can improve performance over ODT with sufficient data. However, unconstrained CART uses unnecessary splits and can result in trees that are orders of magnitude larger than ODTs. Depth-constrained CART may fail to recover an accurate tree even with large training sets and the difference with ODTs does not diminish. Therefore, for both depth-constrained and unconstrained CART, we find that they remain different from ODTs with more data.

7

Model Complexity and Training Data

Claim. The accuracy of greedy trees remains stable when the data size increases linearly with concept complexity (Murthy and Salzberg, 1995).

To test Claim 4, we repeat the experiment by Murthy and Salzberg (1995) by using synthetic data with the number of training samples linear in terms of the number of leaf nodes of the ground truth tree. We set this number to 50 times the number of leaf nodes. Unlike Murthy and Salzberg (1995), we prune the greedy tree and compare it with the optimal tree result instead of comparing it with the ground truth tree.

Fig. 7.17 shows the effect of linearly increasing the sample size with the true tree complexity. It shows that Greedy's True Discovery Rate (and False Discovery Rate) decrease (increase) faster than the ODT's. Interestingly, both methods find trees that have roughly the same number of leaf nodes as the true tree. Regarding Claim 4, both CART with and without a depth limit have a strong decrease in accuracy as the ground truth complexity increases, whereas the ODT's performance remains

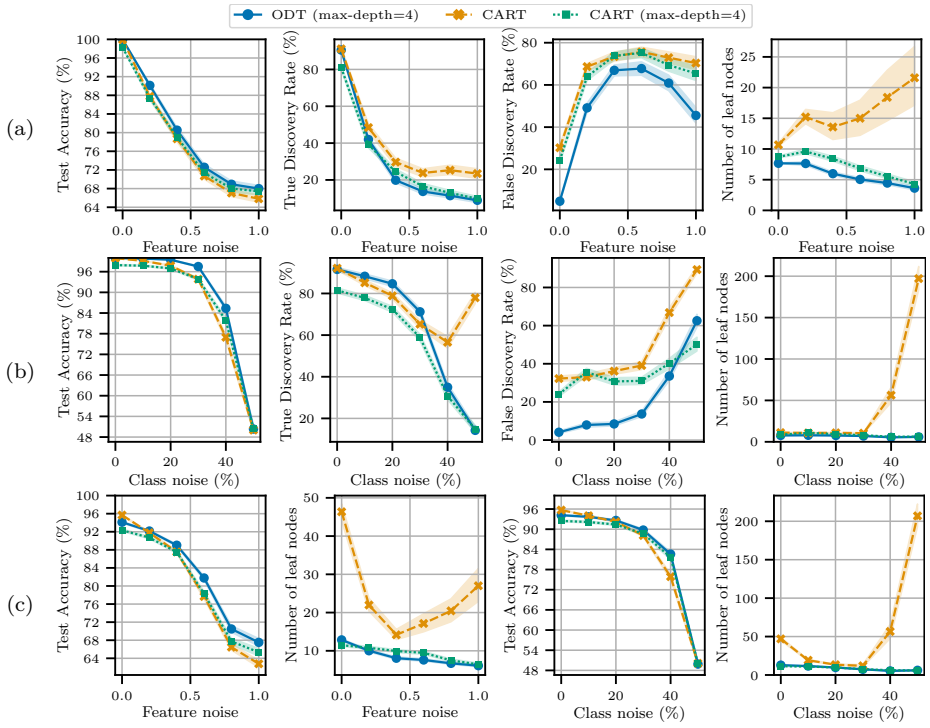


Figure 7.18: Testing Claim 5 on the synthetic tree data for increasing (a) feature noise, and (b) class noise; and (c) on the synthetic linear data for increasing feature and class noise.

close to 100%. Therefore, our results falsify Claim 4. The performance of greedy methods reduces when the true depth increases since increasing true depth requires an increasing number of correct greedy decisions. Since greedy decisions cannot be undone, the probability of making wrong decisions increases.

Overfitting

Claim. Optimal trees are more likely to overfit than greedy trees (Dietterich, 1995).

We addressed overfitting before when observing the results in Fig. 7.15. These results showed that without hyperparameter tuning, ODTs are more prone to overfitting than greedy approaches when data is sparse. However, with tuning and with the same size constraint, we observe that ODTs perform better than greedy trees on average. We further analyze the risk of overfitting by comparing ODTs with greedy on the synthetic data with noise.

Fig. 7.18a shows how both approaches respond to increasing feature noise. For all amounts of feature noise, ODT obtains both a higher test accuracy and smaller trees than both CART approaches. The TDR is mostly similar, except for large amount of feature noise which causes the unconstrained CART to train larger trees which results in a higher TDR, but not in higher accuracy. ODT's FDR is consistently better than CART's.

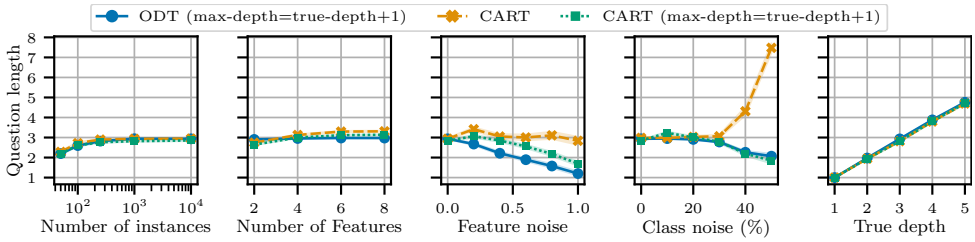


Figure 7.19: The question length on the synthetic data is almost always the same, except for datasets with a strong noise component, mostly supporting Claim 6.

Fig. 7.18b shows that for increasing amounts of class noise, ODT's test accuracy is again consistently higher than both CART approaches. For large amounts of class noise, unconstrained CART obtains a lower test accuracy than both other approaches and also yields significantly larger trees.

For the synthetic linear data, Fig. 7.18c shows similar results. In both cases, with little noise, unconstrained CART achieves a higher accuracy but with a much larger tree. However, when either type of noise increases, ODT's test accuracy becomes higher.

These results show that ODTs are not more sensitive to noise than greedy trees. Combined with the previous result on learning trees with a small training sample, this proves that Claim 5 is false when ODTs are properly tuned. Without tuning, given a fixed size limit, ODTs can overfit more than greedy trees. However, ODTs achieve the same performance as greedy trees at smaller size limits, which means there is no difference in overfitting after tuning. Therefore, the regularization strength of a fixed size on optimal and greedy trees is different and cannot be directly compared.

Question Length

Claim. The question length of greedy trees remains (in practice) close to the optimal tree depth (Goodman and Smyth, 1988; Murthy and Salzberg, 1995).

The question length is the expected number of branching nodes visited by a test instance. We evaluate Claim 6 about the question length of greedy trees on the synthetic data.

Fig. 7.19 shows that the question length of depth-constrained CART remains similar to the optimal approach. Also without a depth limit, CART's question length in almost all cases –for varying training data size, number of features, and ground truth complexity– remains the same as the optimal approach. However, with much feature or class noise, CART's question length is (much) larger than ODT's. This difference is most pronounced for more than 30% class noise. Because Murthy and Salzberg (1995) tested with class noise up to 25% this effect was not previously observed.

Theoretically it can be shown that CART's question length can be arbitrarily larger than optimal (Garey and Graham, 1974). But Claim 6 concerns practical use cases. Therefore, since the difference between CART and ODTs only arises for large noise, we consider Claim 6 supported by our experiments.

	depth = 2					depth = 3					depth = 4				
Samples	10^2	10^3	10^4	10^5	10^6	10^2	10^3	10^4	10^5	10^6	10^2	10^3	10^4	10^5	10^6
50 features	s	s	s	s	s	s	s	s	s	s	s	s	s	s	m
100	s	s	s	s	s	s	s	s	s	s	s	s	s	s	m
150	s	s	s	s	s	s	s	s	s	m	s	s	s	m	m
200	s	s	s	s	s	s	s	s	s	m	s	s	s	m	-
250	s	s	s	s	m	s	s	s	s	m	s	m	m	m	-
300	s	s	s	s	m	s	s	s	m	m	s	m	m	-	-

Table 7.3: Approximate magnitudes of runtimes for training ODTs using STreeD on synthetic data. *s* for (sub)seconds, *m* for minutes, and dashes for runtimes over two hours.

7.4.5. Scalability of Optimal Decision Trees

A final major difference between optimal and greedy approaches is their scalability. The worst-case runtime of dynamic programming ODT methods grows exponentially with the size of the tree, linearly with the number of samples, and exponentially with the number of binary features. Contrasting this with greedy methods whose runtime only grows linearly with the size of the tree, linearly with the number of features, and log-linearly with the number of samples, it is clear that greedy methods scale better in runtime.

Since scalability is one of the limitations of ODTs, we test for what problem sizes the use of ODTs is practically feasible. Table 7.3 provides an overview of runtimes for the optimal method STreeD in terms of seconds, minutes, and hours when trained on synthetically generated data from a random decision tree. We find that training ODTs up to depth four remains practically feasible for datasets up to approximately 250 binary features for 100,000 instances and 150 binary features for one million instances.

We recollect the observation made before that ODTs are specifically good

1. for training small trees up to depth four (Claim 1);
2. for obtaining trees with a good accuracy-interpretability trade-off (Claim 2);
3. and increasingly so for datasets with many instances (Claim 3).

Therefore, we conclude that the best application of ODTs is finding small interpretable trees for medium to large datasets with relatively few but meaningful and well-prepared features.

7.5. Conclusion

We experimentally evaluated many different approaches on how to train optimal decision trees (including several novel ones) and how they compare to greedy approaches. Based on the obtained insights, we provide recommendations for training and evaluating optimal and greedy methods (Recommendations 1-3).

Since the design of greedy top-down induction (TDI) methods prevents direct accuracy optimization, the literature shows a large variety of learning objectives. We identified and analyzed nine of these objectives and observed that the concavity of these objectives leads to counter-intuitive results. Based on this analysis, we

evaluate two new non-concave objective functions. By optimizing ODTs for in-sample accuracy, we found significant improvements in out-of-sample accuracy over objectives such as entropy. For noisy datasets, we show that objectives that have an additional regularizing effect (such as C4.5's pessimistic binomial error) may improve out-of-sample performance, while also improving the sparsity of the trees.

We further introduced two novel methods for tuning a tree's complexity and evaluated these against four existing methods across 180 datasets with trees up to depth four, and 157 datasets with trees up to depth five. The experiments show that tuning has a statistically significant impact on the accuracy. The differences in accuracy between the six methods are small, but they may yield smaller or larger trees. Tuning only the depth is promising if runtime performance is a concern. Tests with synthetic noisy data shows that tuning the question length or the minimum support is significantly worse than the other approaches.

Another important contribution of this work is establishing a set of best practices and the supporting analysis framework for comparing ODT and greedy methods. For this, we analyzed previous comparisons and then used the newly established best experimental practices (Recommendations 2) to test six claims from the literature, leading to the following insights. Our results confirm that (1) ODTs, on average, outperform greedy trees by 1-2% in out-of-sample accuracy for trees trained up to depth four. (2) ODTs, on average, have a better accuracy-interpretability trade-off, which we evaluate with a new metric: the size-weighted accuracy. We refute the claim that (3) differences between greedy and optimal diminish for more data. Depth-constrained greedy methods may fail to recover the true tree, even for large datasets, where ODTs succeed. Unlike previously claimed, (4) greedy methods do not maintain a similar accuracy when the dataset size increases linearly with the true tree complexity. ODTs, on the other hand, do. When the complexity of a tree is properly tuned, we find no support for the claim that (5) ODTs are more prone to overfitting than greedy trees. Finally, we find supporting evidence for the claim that (6) the question length of greedy trees remains close to that of the optimal tree.

Although unrestricted greedy trees can outperform depth-limited ODTs in accuracy, they can quickly grow so large that they cannot be directly interpreted anymore. Random forests or neural networks already suffice if accuracy is the only concern. However, ODTs are an ideal candidate if interpretability is required, as they achieve a superior accuracy-interpretability trade-off over (unrestricted) greedy trees.

Future research could investigate the comparison of ODTs with greedy methods without explicit binarization as a preprocessing step.

Appendices for Chapter 7

7.A. Splitting and Pruning Criteria as ODT Objectives

Greedy top-down induction (TDI) methods typically consider two types of criteria: a local *splitting criterion* that decides how to split a node and a *pruning criterion* that decides which nodes to keep during pruning. Some greedy methods employ another criterion for pruning such as the minimum description length (Mehta et al., 1995) or the pessimistic error rate in C4.5 (Quinlan, 1993b). In this appendix, we list nine such criteria and rewrite them as ODT objectives.

For brevity, we consider only the binary classification case. For all functions, terms that divide by zero (e.g., e/n when $n = 0$) or take the log of zero (e.g., $\log_2(e/n)$ when $e = 0$) are evaluated as zero. We leave out some splitting criteria, such as the twoing rule in CART, if they are not additive or cannot easily be expressed as an objective function for the leaf.

Accuracy: The most direct way to optimize out-of-sample accuracy is to optimize the in-sample accuracy by minimizing the *misclassification score*. The leaf node objective in terms of the leaf node size n and the number of misclassifications e is $f(n, e) = e$. A possible disadvantage of this objective is its 'flatness': it cannot distinguish between many different solutions. For example, $f(4, 2) + f(6, 0) = f(5, 1) + f(5, 1)$.

Gini impurity: Commonly used in TDI heuristics is the Gini impurity. Weighted Gini impurity scores are obtained by multiplying the Gini impurity by the number of instances in that leaf node. Let $p_0 = \frac{e}{n}$ denote the probability of the first class and $p_1 = \frac{n-e}{n}$ the probability of the second class. Then the objective value is

$$f(n, e) = n(1 - p_0^2 - p_1^2)\,.$$

Square root Gini: Kearns and Mansour, 1996 propose to use the square root of the Gini impurity to improve performance on unbalanced datasets:

$$f(n, e) = n\sqrt{1 - p_0^2 - p_1^2}\,.$$

Entropy: The second commonly used TDI criterion is entropy (or information gain). Weighted scores are again obtained by multiplying by the size of the leaf node:

$$f(n, e) = -\frac{n}{2}(p_0 \log_2 p_0 + p_1 \log_2 p_1)\,.$$

Gini impurity and entropy can be expressed as a parameterization of Tsallis entropy (Tsallis, 1988; Wang and Xia, 2017).

Minimum error: Niblett (1987) estimates the expected error for nodes by assuming that every class has equal probability. It depends on the number of labels $|\mathcal{K}|$, and the count of the majority label n_c. In binary classification $|\mathcal{K}| = 2$ and $n_c = n - e$. Therefore, the expected error is

$$n\frac{n - n_c + |\mathcal{K}| - 1}{n + |\mathcal{K}|} = \frac{n(e+1)}{n+2}\,.$$

This is equivalent to Laplace smoothing with a smoothing parameter set to one (add-one smoothing) (Flach, 2012). According to Mingers (1989a), the equal-probability assumption of this approach becomes problematic for a large number of classes.

Pessimistic error: Quinlan (1987) proposed a pessimistic error rate by computing a bound on the expected error rate, which in effect raises the error rate at every leaf by 0.5:

$$f(n,e) = e + \frac{1}{2}\,.$$

This method is similar to a complexity cost per node of 0.5. Since we cover complexity costs in the next section, we do not consider the pessimistic error in this section.

Binomial pessimistic error (Binom.): The commonly used C4.5 method (Quinlan, 1993b) uses an advanced form of pessimistic error by considering a leaf with n training instances and e misclassifications as a 'sample' from a binomial distribution with an unknown misclassifying probability. Since this probability cannot be computed directly, the upper confidence bound of the posterior distribution of this probability, based on a confidence level α, is used as the error rate of the leaf node. The confidence interval width z_α is the z-value from the normal distribution for confidence level α. Let $e' = e + \frac{1}{2}$ be the pessimistic error. Then the binomial pessimistic error can be expressed as:

$$f(n,e) = \begin{cases} n \cdot \left(1 - \exp^{\ln(\alpha)/n}\right) & \text{if } e = 0 \\ e & \text{if } e = n \\ \dfrac{e' + \frac{z_\alpha^2}{2} + \sqrt{z_\alpha^2\left(e'\left(1 - \frac{e'}{n}\right) + \frac{z_\alpha^2}{4}\right)}}{n + z_\alpha^2} \cdot n & \text{otherwise.} \end{cases}$$

In our experiments, we use the same default $\alpha = 0.25$ as C4.5.

Minimum description length (Quinlan): The minimum description length approach states that the best model can be described with the least amount of bits of information because the description length of a model can directly be linked to the posterior probability of a model (Rissanen, 1978; Li and Vitányi, 2008). In practice, the encoding typically consists of two parts: first the encoding of the model and then the encoding of the data that deviates from the model.

Quinlan and Rivest (1989) observe that the cost of encoding a binary string of length n with e ones and $n - e$ zeros can be computed by first encoding the

size n of the string, and then the positions of the e ones, with b representing an upper bound on the number of ones that can occur:

$$L(n, e, b) = \ln(b+1) + \ln \binom{n}{e}. \tag{7.2}$$

Then, for every leaf node with n instances and e misclassifications, encode a bit string that specifies the misclassifications with cost $L(n, e, \lfloor n/2 \rfloor)$ for binary classification:

$$f(n, e) = L(n, e, \lfloor (n+1)/2 \rfloor).$$

The encoding of the branching feature and the leaf node label are part of the tree complexity cost which we cover in the next section.

Minimum description length (Mehta): Mehta et al. (1995) observe that the coding length for Eq. (7.2) may be sub-optimal when e is close to zero, and therefore propose to use stochastic complexity (Krichevsky and Trofimov, 1981; Rissanen, 1997). For binary classification, their formula can be rewritten to

$$f(n, e) = e \ln \frac{n}{e} + (n-e) \ln \frac{n}{n-e} + \frac{1}{2} \ln \frac{n}{2} + \ln \pi.$$

Bayesian: Decision trees are also commonly trained using a Bayesian approach (Chipman et al., 1998; Denison et al., 1998). These approaches find the maximum likelihood tree given some priors. We present the objective function used in the recent work by Sullivan et al. (2024). For the binary case, they assume that each leaf node can be represented by a Bernoulli distribution with parameter $\theta \in [0, 1]$. They assume $\theta \in \text{Beta}(\rho_0, \rho_1)$, the Beta distribution with parameters $\rho_0, \rho_1 \in \mathbb{R}^+$. The values ρ_0 and ρ_1 are hyperparameters, but they fix these values to $\rho_0 = \rho_1 = 2.5$. The error can then be expressed in terms of the Beta function B as follows:

$$f(n, e) = \frac{B(e + \rho_0, n - e + \rho_1)}{B(\rho_0, \rho_1)}.$$

Sullivan et al. (2024) also add a cost based on the priors, which can be considered a complexity cost, and therefore we leave it out in the discussion in this section.

Almost all previous ODT methods optimize the accuracy, but some have considered other objectives, such as balanced accuracy (Lin et al., 2020) or F1-score (Demirović and Stuckey, 2021; Van der Linden et al., 2023). Nijssen and Fromont (2010) have implemented the pessimistic error objective from C4.5 and a Bayesian objective function, but do not discuss their effect on the out-of-sample performance.

7.B. ODT Tuning Approaches

This appendix provides further details on the ODT hyperparameter tuning approaches, the choice of hyperparameters when limited to k options, and more experiments.

7.B.1. Details on Existing Tuning Approaches

We evaluate four existing ODT hyperparameter tuning approaches: depth, size, cost-complexity, and minimum support. Here, we provide some background on each of the tuning methods, and how we select the k options for the hyperparameter. For each approach, we always include a setting that provides no limitation on the tree size. The other settings are derived using linear or log-equal distances between settings. The following provides more details per method:

Depth: A common metric for tree complexity is its *depth*. DL8.5 (Aglin et al., 2020a,b), for example, tunes the depth of the tree. A possible disadvantage is that the number of nodes increases exponentially with respect to the maximum depth, thus providing only a coarse control of the tree size.

In our experiments, we tune the depth parameter $d \in \{0, ..., \text{max-depth}\}$ with equal linear space between the options and with a max-depth of either four or five.

Size: The *number of nodes* can be tuned directly as a hard constraint. Since binary trees always have one more leaf node than branching nodes, the total number of nodes can be counted by the number of branching or leaf nodes alone. This approach is taken by, e.g., MurTree (Demirović et al., 2022) which directly tunes both the maximum depth and the number of branching nodes.

In our experiments, we tune the number of branching nodes $n \in \{0, ..., 2^{\text{max-depth}} - 1\}$. The maximum value is always included and the other $k-1$ options are selected using equal log spacing within this range.

Complexity cost: The most common approach in greedy trees is to penalize the cost of adding a node by a factor λ. This approach is used in most MIP methods (Bertsimas and Dunn, 2017) and also in the optimal sparse approaches such as GOSDT (Lin et al., 2020). Complexity-cost tuning minimizes $\lambda|\mathcal{D}||L| + \sum_{(n,e)\in L} f(n,e)$, with L the set of leaves, $|\mathcal{D}|$ the size of the dataset, and λ the complexity-cost parameter.

The minimum change in λ that may result in a different tree is $\lambda = \frac{1}{|\mathcal{D}| \text{ max-depth}}$ because below this value it is never worth adding nodes to increase accuracy. We set the maximum value to 0.05 and select $k-1$ options from this range using equal log spacing. The minimum step size between values is set to $\frac{1}{|\mathcal{D}| \text{ max-depth}}$. In all cases, we add $\lambda = 0$.

Note that Lin et al. (2020) recommend for their method GOSDT to use $\lambda \geq \frac{1}{|\mathcal{D}|}$ for faster training, but this setting can exclude larger trees that are more accurate. Additionally, in their experiments, they aim to acquire trees of at most n leaves by setting $\lambda = \frac{1}{n}$. However, this also filters out many trees that have much less than n leaves, since generally a single leaf node already has an accuracy greater than 0.5 for binary classification, so that even a perfect tree with n leaves given such λ would have a lower score than a single leaf node. Therefore, both of these settings are too conservative.

Minimum support: The *minimum support* is a hard constraint on the minimum number of instances that in a leaf node. Minimum support is more common in the data mining literature, e.g., Nijssen and Fromont (2007). It is also introduced as a soft constraint (Vilas Boas et al., 2021).

In our experiments, we tune the minimum leaf node size m as a percentage of the dataset size $|\mathcal{D}|$. The minimum value for m is $\frac{1}{|\mathcal{D}|}$: precisely one sample should end up in each leaf node. For the maximum value, we compute the frequency of the majority class $|\mathcal{D}_{\text{majority}}|$ and set the maximum value of m to $1 - \frac{|\mathcal{D}_{\text{majority}}|}{|\mathcal{D}|}$. Any value higher than this will always result in a single leaf node. We obtain k values from this range using equal log spacing, with a minimum step size of $\frac{1}{|\mathcal{D}|}$.

We leave out selecting a tree structure from a fixed set (Günlük et al., 2021), because this approach was chosen specifically to deal with the scalability limitations of MIP.

7.B.2. Details on the New Tuning Approaches

We set up the two new tuning approaches introduced in the main text as follows:

Question length: In our experiments, we select $k-1$ options for ω using equal log spacing from the interval $[\frac{1}{|\mathcal{D}| \text{ max-depth}}, 0.1]$. Additionally, we always add the option $\omega = 0$. We also use $\frac{1}{|\mathcal{D}| \text{ max-depth}}$ as the minimum step size.

Smoothing: In our experiments, we select $k-1$ options for x using equal log spacing from the interval $[\frac{1}{\text{max-depth}}, 0.05|\mathcal{D}|]$. Additionally, we always add the option $x = 0$. We also use $\frac{1}{\text{max-depth}}$ as the minimum step size.

7.B.3. Extended Tuning Experiments

Here, we report additional experiments on the ODT tuning methods. We test the performance of the methods for other values of k, the number of hyperparameter options, and we report the average runtime for each approach. While in the main text, we experimented with $k = 16$, here we test for $k = 5, 10, 20$. For runtime comparison, we ran all experiments on an Intel Xeon E5-6248R 3.0GHz with 100GB RAM using one thread.

Fig. 7.20 shows that the choice of k has no major impact on the out-of-sample performance of the tuning methods. In almost all cases, running with more hyperparameter options results in a lower average number of leaf nodes. As expected, increasing k by a factor two, also increases the runtime for each method by a factor of roughly two.

These results also show that the runtimes of all methods are in the same order of magnitude. The only exception to this is tuning the depth. Therefore, the choice of hyperparameter tuning and the number of parameter options to test mostly depends on how much time one is willing to spend on finding *small* trees and less important for optimizing accuracy.

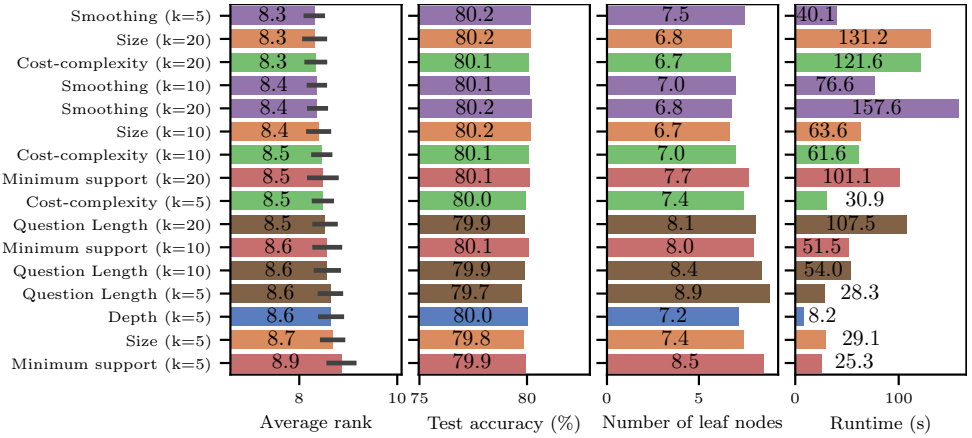


Figure 7.20: Performance of all tuning methods for $k = 5, 10, 20$ with max-depth = 4 on the 180 OpenML datasets. The color indicates the tuning method. Test accuracy is roughly the same for all methods. Increasing k typically yields smaller trees. As expected, runtime scales roughly linear with k.

7.C. Previous Comparisons between Optimal and CART

This section reviews previous comparisons between optimal and greedy decision tree learning methods, each different in its experiment design and sometimes in its conclusion. We restrict our review to binary axis-aligned trees. First, we summarize the comparisons from papers that proposed ODT methods. Second, we discuss other papers that compare greedy heuristics and optimal methods. Finally, we discuss the best and poor comparison practices as input for a fair comparison method presented after. We highlight the encountered claims that were mentioned above.

7.C.1. Comparisons in ODT Papers

Papers that propose new ODT methods typically aim to train a decision tree with a given size constraint that achieves the best out-of-sample performance. Nijssen and Fromont (2007, 2010) compare their optimal DL8 algorithm with J48, an implementation of C4.5. When trained on the same discretized data, without a depth limit, but with the same minimum support constraint, DL8 is significantly better for 9 out of 20 datasets and worse for one, while yielding trees that are 1.5 times larger than J48. However, when J48 is trained without the minimum support constraint and with the non-discretized data, J48 outperforms DL8 on out-of-sample accuracy for most datasets.

Bertsimas and Dunn (2017) develop the optimal MIP method OCT and compare it with CART on real and synthetic data. When CART is constrained to the same depth limit as OCT (up to depth four), they conclude that OCT, on average, has a statistically significant 1-2% better out-of-sample accuracy (Claim 1). The largest

difference is observed at depth two. They hypothesize that the smaller difference for depths three and four is the result of OCT not reaching optimality within their time limit. When CART is run with a depth limit of ten, it is negligibly better than OCT at depth four.

The main restriction of their analysis is the scalability of OCT. Because of this, they restrict synthetic data analysis to datasets with only 100 instances and two continuous features. They also experiment with datasets up to 1600 instances, but only on ground truth trees of depth two. When training OCT on the synthetic data, they set the maximum depth to the true depth, which prevents overfitting.

Verwer and Zhang (2019) compare the optimal MIP methods BinOCT, DTIP (Verwer and Zhang, 2017), and OCT with depth-constrained CART on datasets with a few thousand instances. They report results without hyperparameter tuning and observe that the ODTs are significantly better for depths two and three and slightly better for depth four (Claim 1).

Lin et al. (2020) propose GOSDT, an ODT method with a sparsity coefficient. They conclude that GOSDT obtains a better accuracy-interpretability trade-off than other methods, including CART (Claim 2). This is based on an experiment on six small datasets with a coarse binarization applied to both GOSDT and CART. CART is tuned using the maximum depth parameter (from one to six), instead of tuning the complexity-cost as is normally done. They tune GOSDT using complexity-cost tuning without a depth limit.

Demirović et al. (2022) compare their optimal MurTree algorithm and CART on binarized datasets with up to 43500 instances and 1163 binary features. They run MurTree for different depths (from one to four) and number of nodes, and CART for different depths (from one to four), and report the best test accuracy for each method. They too report an average out-of-sample improvement of 1-2% over CART (Claim 1).

The same trend appears in other papers that propose new ODT methods. The aim is to show ODTs' superior performance under a fixed depth limit. An exception is (Alès et al., 2024), who compare with unconstrained greedy approaches. Results are often shown for fixed hyperparameters (Hu et al., 2020; Mazumder et al., 2022; Liu et al., 2024). Scalability limits the analysis to small datasets (Hu et al., 2020; Günlük et al., 2021; Alès et al., 2024) or larger datasets are run only with a maximum depth of two (Hua et al., 2022).

7.C.2. Other Comparisons

Papers that do not propose new ODT methods typically have another aim: the best out-of-sample accuracy without imposing depth constraints on the tree.

Murthy and Salzberg (1995) compare the greedy approach to known true (optimal) trees on a synthetic benchmark. They observe that the greedy tree is approximately one standard deviation larger than the true tree size while the question length (what they call expected depth) is very close to optimal, which was also observed by Goodman and Smyth (1988) (Claim 6). However, the maximum depth of greedy trees is on average approximately two times higher than the true depth. When they increase the dataset size linearly with the true tree complexity, they observe almost

no drop in accuracy for the greedy approach (Claim 4). From this result, Costa and Pedreira (2023) hypothesize that the gap between optimal and greedy approaches diminishes for more data (Claim 3).

Dietterich (1995) concludes from the empirical results by Quinlan and Cameron-Jones (1995) that optimal methods are more prone to overfitting (Claim 5). Exhaustively searching through all possible models may yield smaller models, but is also more prone to finding small patterns that do not represent the ground truth. Therefore Dietterich concludes that it is better to train greedy methods with a model complexity penalty.

Zharmagambetov et al. (2021) compare greedy methods with their local search method TAO (Carreira-Perpinán and Tavallali, 2018) and the optimal methods OCT and GOSDT. They conclude that most methods perform similarly, except TAO, which outperforms the other methods. In many cases, CART outperforms OCT and GOSDT. For CART and TAO, they train greedy trees up to depth 30. For OCT, they use the results reported in (Bertsimas and Dunn, 2017), which go up to depth four. They train GOSDT with a high complexity cost, yielding trees that are on average no larger than 3.4 leaf nodes for any of the datasets.

Sullivan et al. (2024) propose MAPTree, a search algorithm that finds the maximum a posteriori tree. They compare with DL8.5, GOSDT, and CART and conclude that MAPTree outperforms these approaches, which leads them to question the 'optimality' of ODTs. They observe that DL8.5 is prone to overfitting, while GOSDT is prone to underfitting, and both are sensitive to hyperparameter tuning, whereas MAPTree is not. However, these results are from averaging the performance per hyperparameter setting over all datasets, rather than tuning the hyperparameters for each individual run. Additionally, they evaluate MAPTree without a depth limit, while other methods (including CART) are run with a depth limit.

Marton et al. (2024) learn axis-aligned trees with gradient descent and compare the results a.o. to CART and DL8.5. GradTree outperforms the other methods for binary classification, while CART performs the best for multi-class classification. The ODT approach DL8.5 performs the second worst in both cases: only the evolutionary approach is worse. They tune each method using random search, except for DL8.5 which they fix to a maximum depth of four. The other methods were typically trained up to depths 7-10.

7.D. Additional Experiments

Section 7.D.1 provides more details on the train and test accuracy of various objectives. Section 7.D.2 measures the effect of binarization for ODTs.

7.D.1. Train Accuracy

Fig. 7.21 and 7.22 show the train and test accuracies obtained by training optimal decision trees for three objectives for an increasing node limit for small and medium-sized datasets. All trees are trained with max-depth = 4. The training accuracy of the three objectives is typically close. For the larger datasets, Gini impurity obtains a lower training accuracy. Optimizing accuracy or M-loss results in (almost) the

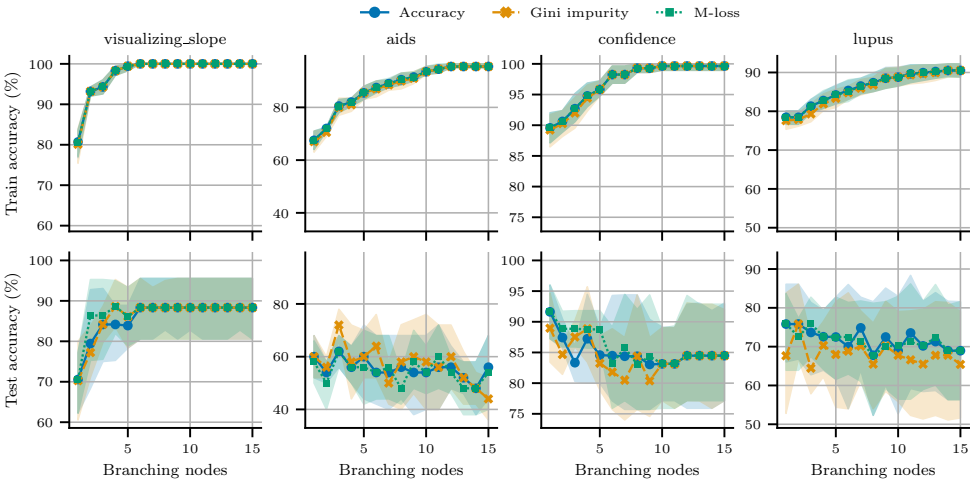


Figure 7.21: Train and test accuracy for ODTs with different objectives for increasing size limits for the datasets visualizing_slope (n=44), aids (n=50), confidence (n=72), and lupus (n=87). Train accuracy is always close. Test accuracy differences are larger.

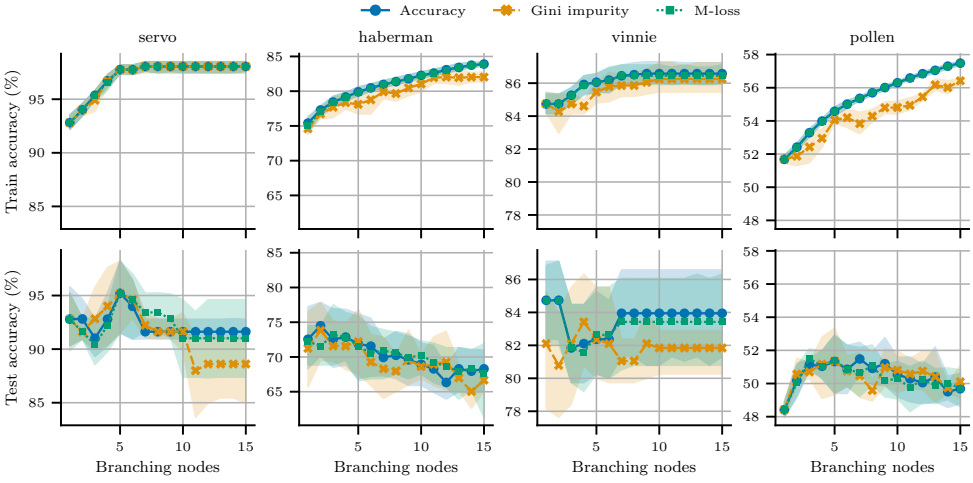


Figure 7.22: Train and test accuracy for ODTs with different objectives for increasing size limits for the datasets servo (n=167), haberman (n=306), vinnie (n=380), and pollen (n=3848). For haberman, vinnie, and pollen, the train accuracy of Gini impurity is worse.

same training accuracy.

For several of these datasets, the training accuracy plateaus early. For some (other) datasets, the test accuracy remains stable or decreases when trained with more nodes, indicating that the trees are possibly overfitting. The variance in the test accuracy results is large, making it hard to draw conclusions from these figures.

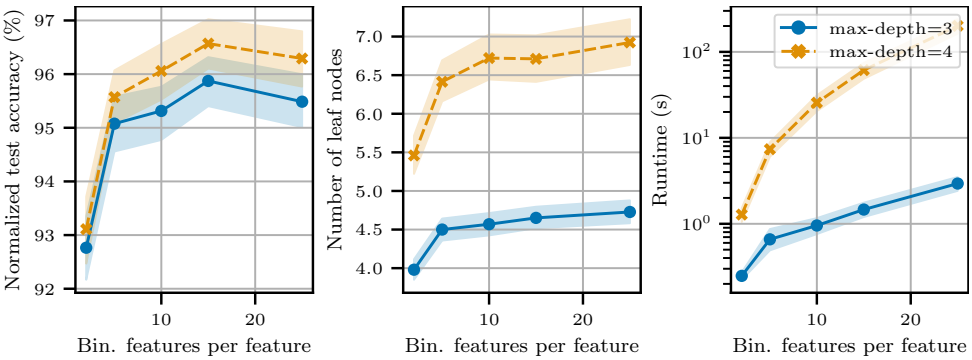


Figure 7.23: The normalized accuracy for an increasing number of binary features per original feature. At some point, more binary features add little extra information, while significantly increasing the runtime.

7.D.2. Binarization

To test the effect of binarizing features, we experiment with changing the number of binary features per original numeric or categorical feature. For all datasets, we compute optimal trees of maximum depth three or four when creating 2, 5, 10, 15, or 25 binary features per original feature. We then measure the test accuracy for each tree and compute the normalized test accuracy by dividing it by the highest test accuracy obtained for that dataset. Fig. 7.23 shows how on average the test accuracy plateaus when creating more and more binary features per original feature. Our selection of 10 binary features per original feature on average is 0.6% less accurate for max-depth = 3 and 0.5% less accurate for max-depth = 4 than when using 15 binary features per original feature. Fig. 7.23 also shows the exponential increase in runtime when the number of binary features is increased.

What binarization technique works best is a topic that we leave aside in this chapter. Other work studies this, such as McTavish et al. (2022) who use ensembles for binarization, and Piccialli et al. (2024) who use counterfactuals for discretization. Recently, also new ODT methods have been proposed that do not require explicit binarization (Mazumder et al., 2022; Shati et al., 2023b; Brița et al., 2025).

In this chapter, the precise method of binarization is not a major concern. We assume that the binarization is given, so that all methods work on precisely the same data. In this way, we eliminate this difference between methods and can focus in the comparison on the difference between the greedy and optimal approaches.

7.E. Datasets

Table 7.4 lists the 180 OpenML datasets used in the experiments in this chapter (Vanschoren et al., 2013; Feurer et al., 2021).

ID	Dataset	Samples	Features	Binarized features	Class imbalance
919	rabe_166	40	1	10	0.53
857	bolts	40	7	31	0.65
938	sleuth_ex1221	42	9	90	0.55
40660	analcatdata_fraud	42	11	19	0.69
791	diabetes_numeric	43	2	20	0.60
729	visualizing_slope	44	3	30	0.61
777	sleuth_ex1714	47	7	70	0.57
835	analcatdata_vehicle	48	4	9	0.56
817	diggle_table_a1	48	4	40	0.52
942	chscase_health	50	3	27	0.52
346	aids	50	4	30	0.50
787	witmer_census_1980	50	4	40	0.52
476	analcatdata_bankruptcy	50	5	50	0.50
892	sleuth_case1201	50	6	60	0.52
713	vineyard	52	2	18	0.54
467	analcatdata_japansolvent	52	8	80	0.52
790	elusage	55	2	20	0.56
864	sleuth_ex2015	60	7	61	0.55
887	mbagrade	61	2	11	0.52
755	sleuth_ex1605	62	5	50	0.50
804	hutsof99_logis	70	7	37	0.51
1015	confidence	72	3	30	0.83
893	visualizing_hamster	73	5	50	0.55
859	analcatdata_gviolence	74	8	51	0.58
945	kidney	76	6	36	0.53
459	analcatdata_asbestos	83	3	14	0.55
472	lupus	87	3	26	0.60
862	sleuth_ex2016	87	10	80	0.52
946	visualizing_ethanol	88	2	14	0.51
40683	postoperative-patient-dat	88	8	22	0.73
1055	cm1_req	89	8	17	0.78
479	analcatdata_cyyoung9302	92	9	67	0.79
891	sleuth_case1202	93	6	44	0.61
731	baskball	96	4	40	0.51
465	analcatdata_cyyoung8092	97	10	78	0.75
865	analcatdata_neavote	100	2	10	0.93
875	analcatdata_chlamydia	100	3	16	0.81
1463	blogger	100	5	13	0.68
754	fri_c0_100_5	100	5	50	0.54
46544	dataset_analcatdata_credi	100	6	36	0.73
965	zoo	101	16	20	0.59
45615	appendicitis_test_edsa	106	7	70	0.80
771	analcatdata_michiganacc	108	3	24	0.56
890	cloud	108	7	61	0.70
736	visualizing_environmental	111	3	30	0.52
448	analcatdata_boxing1	120	3	21	0.65
1556	acute-inflammations	120	6	15	0.51

ID	Dataset	Samples	Features	Binarized features	Class imbalance
714	fruitfly	125	4	24	0.61
40681	mux6	128	6	6	0.50
924	humandevel	130	1	10	0.50
867	visualizing_livestock	130	2	15	0.81
1075	datatrieve	130	8	71	0.92
885	transplant	131	3	30	0.63
921	analcatdata_seropositive	132	3	23	0.65
974	hayes-roth	132	4	11	0.61
719	veteran	137	7	36	0.69
1013	analcatdata_challenger	138	2	12	0.93
784	newton_hema	140	3	30	0.50
902	sleuth_case2002	147	6	24	0.53
1006	lymph	148	18	47	0.55
969	iris	150	4	39	0.67
955	tae	151	5	30	0.66
1026	grub-damage	155	8	48	0.68
40669	corral	160	6	6	0.56
748	analcatdata_wildcat	163	5	29	0.71
747	servo	167	4	19	0.77
463	backache	180	31	99	0.86
801	chscase_funds	185	2	16	0.53
941	lowbwt	189	9	37	0.52
1012	flags	194	28	117	0.64
446	prnn_crabs	200	7	61	0.50
733	machine_cpu	209	6	49	0.73
796	cpu	209	7	58	0.75
895	chscase_geyser1	222	2	20	0.60
41538	conference_attendance	246	6	24	0.87
464	prnn_synth	250	2	20	0.50
776	fri_c0_250_5	250	5	50	0.50
1495	qualitative-bankruptcy	250	6	18	0.57
811	rmftsa_ctoarrivals	264	2	20	0.62
450	analcatdata_lawsuit	264	4	30	0.93
336	SPECT	267	22	22	0.79
1073	jEdit_4.0_4.2	274	8	66	0.51
23499	breast-cancer-dropped-mis	277	9	37	0.71
1121	badges2	294	10	43	0.71
40710	cleve	303	13	70	0.54
43	haberman	306	3	26	0.74
1524	vertebra-column	310	6	60	0.68
818	diggle_table_a2	310	8	79	0.53
1167	pc1_req	320	8	31	0.67
925	visualizing_galaxy	323	4	36	0.54
1011	ecoli	336	7	52	0.57
1048	jEdit_4.2_4.3	369	8	68	0.55
860	vinnie	380	2	13	0.51
1025	analcatdata_germangss	400	5	16	0.78

ID	Dataset	Samples	Features	Binarized features	Class imbalance
909	chscase_census2	400	7	70	0.51
1511	wholesale-customers	440	8	65	0.68
1498	sa-heart	462	9	79	0.65
814	chscase_vine2	468	2	18	0.55
724	analcatdata_vineyard	468	3	29	0.56
4329	thoracic_surgery	470	16	54	0.85
767	analcatdata_apnea1	475	3	20	0.87
884	fri_c0_500_5	500	5	50	0.50
750	pm10	500	7	70	0.51
886	no2	500	7	70	0.50
40690	threeOf9	512	9	9	0.54
335	monks-problems-3	554	6	15	0.52
333	monks-problems-1	556	6	15	0.50
949	arsenic-female-bladder	559	4	40	0.86
826	sensory	576	11	32	0.59
334	monks-problems-2	601	6	15	0.66
997	balance-scale	625	4	16	0.54
770	strikes	625	6	60	0.50
774	disclosure_x_bias	662	3	30	0.52
827	disclosure_x_noise	662	3	30	0.50
795	disclosure_x_tampered	662	3	30	0.51
931	disclosure_z	662	3	30	0.53
40981	Australian	690	14	80	0.56
1464	blood-transfusion-service	748	4	34	0.76
37	diabetes	768	8	73	0.65
1014	analcatdata_dmft	797	4	20	0.81
44268	anneal	898	38	104	0.54
50	tic-tac-toe	958	9	27	0.65
40693	xd6	973	9	9	0.67
799	fri_c0_1000_5	1000	5	50	0.50
45604	dummy	1000	6	60	0.73
43255	1StudentPerfromance	1000	7	43	0.52
741	rmftsa_sleepdata	1024	2	14	0.50
40702	solar-flare	1066	10	26	0.83
40706	parity5_plus_5	1124	10	10	0.50
934	socmob	1156	5	31	0.78
40680	mofn-3-7-10	1324	10	10	0.78
1462	banknote-authentication	1372	4	40	0.56
983	cmc	1473	9	35	0.57
40646	GAMETES_Epistasis_2-Way_2	1600	20	60	0.50
40649	GAMETES_Heterogeneity_20a	1600	20	58	0.50
991	car	1728	6	21	0.70
962	mfeat-morphological	2000	6	43	0.90
914	balloon	2001	1	10	0.76
772	quake	2178	3	30	0.56
40704	Titanic	2201	3	5	0.68
737	space_ga	3107	6	60	0.50

ID	Dataset	Samples	Features	Binarized features	Class imbalance
44127	phoneme	3172	5	49	0.50
3	kr-vs-kp	3196	36	38	0.52
871	pollen	3848	5	50	0.50
728	analcatdata_supreme	4052	7	24	0.76
720	abalone	4177	8	73	0.50
40983	wilt	4839	5	50	0.95
45039	compas-two-years	4966	11	34	0.50
44160	rl	4970	12	69	0.50
1460	banana	5300	2	20	0.55
803	delta_ailerons	7129	5	50	0.53
43922	mushroom	8124	22	109	0.52
807	kin8nm	8192	8	80	0.51
725	bank8FM	8192	8	73	0.60
816	puma8NH	8192	8	80	0.50
923	visualizing_soil	8641	4	31	0.55
819	delta_elevators	9517	6	47	0.50
44126	bank-marketing	10578	7	51	0.50
4534	PhishingWebsites	11055	30	46	0.56
45060	online_shoppers	12330	17	136	0.85
959	nursery	12960	8	26	0.67
1046	mozilla4	15545	5	40	0.67
44162	compass	16644	17	111	0.50
45558	Pulsar-Dataset-HTRU2	17898	8	80	0.91
45028	california	20634	8	80	0.50
823	houses	20640	8	80	0.57
43904	law-school-admission-bian	20800	10	53	0.68
843	house_8L	22784	8	75	0.70
45037	BitcoinHeist_Ransomware	24780	7	48	0.50
42493	airlines	26969	7	67	0.55
43900	amazon_employee_access	32769	9	90	0.94
44156	electricity	38474	8	68	0.50
137	BNG(tic-tac-toe)	39366	9	27	0.65
43901	click_prediction_small	39926	8	56	0.83
881	mv	40768	10	75	0.60
46554	Loan_Status	45000	13	94	0.78
45547	Cardiovascular-Disease-da	70000	11	49	0.50
45022	Diabetes130US	71090	7	41	0.50
40922	Run_or_walk_information	88588	6	60	0.50

Table 7.4: List of OpenML datasets used in this paper.

Additionally, we perform some experiments on datasets with more than 100,000 samples:

- *covertype* (ID 44121) with 566,602 samples, 10 features, and 100 binarized features.
- *Higgs* (ID 44129) with 940,160 samples, 24 features, and 240 binarized features.

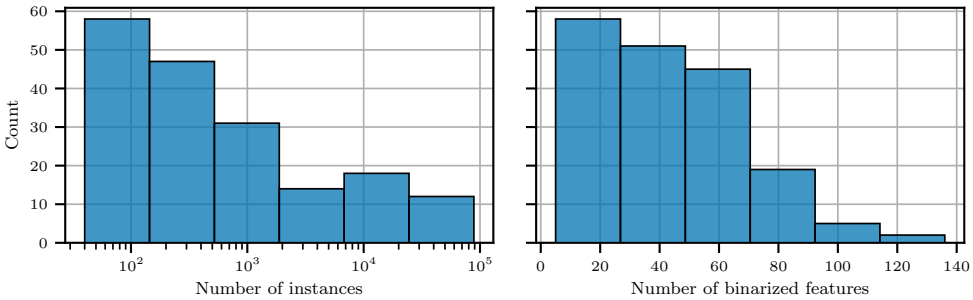


Figure 7.24: Histogram of the number of instances and binarized features in the datasets considered in our experiments.

Fig. 7.24 shows a histogram of the number of instances and the number of binarized features of the datasets considered in this chapter.

8

Scalability for Optimization with Continuous Features

Preface So far in this thesis, we have worked with the common assumption that the feature data is binarized as a preprocessing step. This chapter proposes several algorithmic techniques to improve scalability if we optimize with the numerical feature data directly. This allows to split and test on many more predicates than a coarse binarization allows, resulting in higher train accuracy and, on average, higher test accuracy as well.

Abstract Computing an optimal classification tree that provably maximizes training performance within a given size limit, is NP-hard, and in practice, most state-of-the-art methods do not scale beyond computing optimal trees of depth three. Therefore, most methods rely on a coarse binarization of continuous features to maintain scalability. We propose a novel algorithm that optimizes trees directly on the continuous feature data using dynamic programming with branch-and-bound. We develop new pruning techniques that eliminate many sub-optimal splits in the search when similar to previously computed splits and we provide an efficient subroutine for computing optimal depth-two trees. Our experiments demonstrate that these techniques improve runtime by one or more orders of magnitude over state-of-the-art optimal methods and improve test accuracy by 5% over greedy heuristics.

8.1. Introduction

One major drawback of most specialized DP algorithms for optimal decision trees (ODTs), such as DL8.5 (Aglin et al., 2020a), MurTree (Demirović et al., 2022), and

Parts of this chapter have been published in Brița, Van der Linden, and Demirović (2025), "Optimal Classification Trees for Continuous Feature Data Using Dynamic Programming with Branch-and-Bound", in *Proceedings of AAAI-25*. This paper resulted from Cătălin Brița's bachelor research project, supervised by Emir Demirović and me. Cătălin Brița was responsible for most of the implementation and investigation, and part of the methodology. I did most of the conceptualization, methodology, and writing. For this chapter, the introduction has been shortened and the related work has become part of Chapter 2.

our method proposed in Chapter 4, is that they cannot directly deal with numeric data which are frequently present in real-world datasets. Therefore, these algorithms either use a coarse binarization resulting in loss of optimality; or require a binary feature for every possible threshold on the numeric data, which drastically hurts scalability, because their runtime scales exponentially with the number of such features.

Similarly, among the general purpose solvers, such as MIP and SAT, almost all models require a coarse binarization to improve scalability. While the first MIP approaches for ODTs (Bertsimas and Dunn, 2017; Verwer and Zhang, 2017) both optimized directly on the continuous feature data, many of those that followed use binarization of the continuous feature data to improve the scalability (Verwer and Zhang, 2019; Günlük et al., 2021; Aghaei et al., 2024; Liu et al., 2024). Among SAT methods, all early methods also require binarization of continuous feature data (Narodytska et al., 2018; Avellaneda, 2020; Hu et al., 2020; Janota and Morgado, 2020; Alòs et al., 2023). As far as we know, Shati et al. (2021, 2023b) propose the only SAT-based algorithm that can directly process continuous and categorical features.

To the best of our knowledge, Quant-BnB (Mazumder et al., 2022) is the only specialized optimal algorithm for ODTs that processes continuous features directly. This branch and bound (BnB) algorithm considers splits at certain quantiles of the feature distribution. Obtained solutions are used as bounds for pruning, whereas other parts are further explored with splits on quantiles of subregions of the feature distribution. Though Quant-BnB outperforms other algorithms by a large margin on numeric data, scalability is still an issue: it requires several hours to find trees of depth three for some datasets and the authors recommend against using it beyond depth three.

In this chapter, we present ConTree, which combines existing DP and BnB techniques with new bounding techniques to improve the scalability of computing optimal classification trees for numeric data. Our new lower-bounding techniques prune large parts of the search space while adding negligible computational overhead. Furthermore, for depth-two trees, we propose a specialized subroutine that exploits the fact that we can sort numeric data.

Our experiments show that these algorithmic improvements boost scalability by one or more orders of magnitude over Quant-BnB and three previous MIP and SAT approaches by an ever larger margin. This makes ConTree the first approach to compute depth-four ODTs beyond small or binarized datasets within a reasonable time. When trained with the same size limit, ConTree's test accuracy is on average 5% higher than CART and 0.7% higher than ODTs trained with a coarse binarization.

8.2. Preliminaries

In this section, we introduce notation, formally define the problem, and describe the lower-bounding technique that provides the basis for ConTree's pruning.

Notation Let $\mathcal{D}$ describe a dataset with $n = |\mathcal{D}|$ observations (x, y) with $x \in \mathbb{R}^p$ and $y \in \mathcal{Y}$ describing respectively the feature vector and label of an observation.

Here p is the number of features and $\mathcal{Y}$ the set of class labels. The set of all features is denoted as $\mathcal{F} = \{f_1, \ldots, f_p\}$. Each observation x contains the values of these p features such that x_f is the value of feature f in observation x. Let $\mathcal{D}^f$ denote the sorted values x_f for $(x, y) \in \mathcal{D}$ and let U^f describe all unique sorted values in $\mathcal{D}^f$ (with $\mathcal{D}$ determined by context) and similarly

$$S^f = \left\{ \frac{U_1^f + U_2^f}{2}, ..., \frac{U_{|U^f|-1}^f + U_{|U^f|}^f}{2} \right\}, \tag{8.1}$$

the set of possible thresholds on feature f for $\mathcal{D}$: the midpoints between the unique feature values. Let $m = |S^f|$ be the number of possible thresholds. Given a threshold $\tau \in S^f$, let $z(\tau)$ represent the index of the observation in $\mathcal{D}^f$ with the largest value for f smaller than τ. Similarly, let $u(\tau)$ represent the index of τ in U^f. Finally, let $\mathcal{D}(f \leq \tau)$ describe the subset of observations $(x, y) \in \mathcal{D}$ where $x_f \leq \tau$.

Problem definition A decision tree is a function $t : \mathbb{R}^p \to \mathcal{Y}$ that recursively splits the feature space $\mathbb{R}^p$ into sub-regions and predicts the label of each sub-region. Let $\mathcal{T}(\mathcal{D}, d)$ describe the set of all decision trees for the dataset $\mathcal{D}$ with a maximum depth of d. Then the ODT t_{opt} is the tree within that set that minimizes the misclassification score:

$$t_{\text{opt}} = \underset{t \in \mathcal{T}(\mathcal{D}, d)}{\arg\min} \sum_{(x,y) \in \mathcal{D}} \mathbb{1}(t(x) \neq y). \tag{8.2}$$

This work is limited to binary axis-aligned trees: every branching node splits on precisely one feature $f \in \mathcal{F}$ based on a threshold τ such that every observation with $x_f \leq \tau$ goes left in the tree while the rest goes to the right.

Similarity lower bounding Hu et al. (2019), Lin et al. (2020), and Demirović et al. (2022) propose a similarity-based lower-bounding (SLB) technique that compares a new dataset $\mathcal{D}_{new}$ with a previously analyzed dataset $\mathcal{D}_{old}$ to derive a lower bound on the misclassification score of the new dataset. SLB assumes that all new observations in the new dataset will be classified correctly and all removed observations from the old dataset were classified incorrectly, yielding the following lower bound:

$$\theta_{\mathcal{D}_{new}} \geq \theta_{\mathcal{D}_{old}} - |\mathcal{D}_{old} \setminus \mathcal{D}_{new}|, \tag{8.3}$$

where θ is the misclassification score a decision tree with the same depth limit can achieve on the dataset. From this property, we derive three novel pruning techniques below.

8.3. The ConTree Algorithm

We present the ConTree algorithm (CT) which constructs an ODT by recursively performing splits on every branching node within a full tree of pre-defined depth. Subproblems are identified by the dataset $\mathcal{D}$ and the remaining depth limit d. This

results in the following recursive DP formulation:

$$\mathrm{CT}(\mathcal{D}, d) = \begin{cases} \min\limits_{\hat{y} \in \mathcal{Y}} \sum\limits_{(x,y) \in \mathcal{D}} \mathbb{1}(\hat{y} \neq y) & \text{if } d = 0 \\ \min\limits_{f \in \mathcal{F}} \mathrm{Branch}(\mathcal{D}, d, f) & \text{if } d > 0 \end{cases} \tag{8.4}$$

Leaf nodes assign the label with the least misclassifications. Branching nodes find the feature f with the best misclassification score from the subtrees by calling the subprocedure Branch which iterates over all possible split thresholds τ:

$$\mathrm{Branch}(\mathcal{D}, d, f) = \min_{\tau \in S^f} \mathrm{Split}(\mathcal{D}, d, f, \tau)\,. \tag{8.5}$$

Every split results in two subproblems:

$$\mathrm{Split}(\mathcal{D}, d, f, \tau) = \mathrm{CT}\left(\mathcal{D}(f \leq \tau), d-1\right) + \mathrm{CT}\left(\mathcal{D}(f > \tau), d-1\right)\,. \tag{8.6}$$

Given a splitting feature f, computing the misclassification score θ_τ for all possible split points $\tau \in S^f$ is computationally expensive since each split point considered requires solving two (potentially large) subproblems. Therefore, we provide the following runtime improvements (each of which preserves optimality):

Lower-bound pruning three novel pruning techniques specifically designed for continuous features to speed up the computation without losing optimality;

Depth-two subroutine a subroutine for depth-two trees that iterates over sorted feature data to update class occurrences and efficiently solves the two depth-one subproblems in Eq. (8.6) simultaneously;

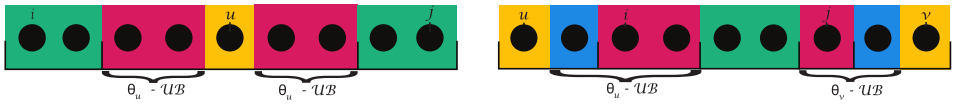
Caching the same dataset caching technique as Demirović et al. (2022): ConTree reuses cached solutions to subproblems (defined by $\mathcal{D}$ and d).

To control the trade-off between training time and accuracy, we also provide a max-gap parameter to set the maximum permissible gap to the optimal solution.

8.3.1. Pruning Techniques

Based on the similarity-based lower bound presented in Eq. (8.3), we present three novel pruning techniques to reduce the number of split points that need to be considered in Eq. (8.5) without losing optimality. The key idea is that if the feature data is sorted, the solution of any Split call with threshold τ provides a lower bound for all next calls with a different threshold τ' since we can easily count how many observations shifted from the left to the right subtree by subtracting their indices in the sorted data: $|z(\tau) - z(\tau')|$.

Theorem 8.3.1. *Let $\mathcal{UB}$ be the best solution so far or the score needed to obtain a better solution. Let $\theta_\tau = \mathrm{Split}(\mathcal{D}, d, f, \tau)$ be the optimal misclassification score for the subtree when branching on f with threshold τ. Then any other threshold τ' with $|z(\tau) - z(\tau')| \leq \theta_\tau - \mathcal{UB}$ cannot yield an improving solution.*



(a) Neighborhood pruning (b) Interval shrinking

Figure 8.1: The split points u and v for which the score θ_u and θ_v are calculated are yellow. The newly pruned values are shown in red. Green indicates the remaining split points for further search. Blue indicates values outside of $[i..j]$ which are unaffected.

Proof. This follows directly from Eq. (8.3). If $\tau' > \tau$, then $|\mathcal{D}(f \leq \tau') \setminus \mathcal{D}(f \leq \tau)| = z(\tau') - z(\tau)$ and if $\tau' < \tau$, then $|\mathcal{D}(f > \tau') \setminus \mathcal{D}(f > \tau)| = z(\tau) - z(\tau')$. Therefore, the SLB for a split at τ' is: $\theta_{\tau'} \geq \theta_\tau - |z(\tau) - z(\tau')| \geq \mathcal{UB}$. □

Corollary 8.3.2. *Let u be a split index with its corresponding solution value θ_u and index $z(u)$ within $\mathcal{D}^f$. Let Δ be the difference between θ_u and $\mathcal{UB}$ and at least one: $\Delta = \max(1, \theta_u - \mathcal{UB})$. Any improving split must have a threshold smaller than the value in $\mathcal{D}^f$ at index $z(u) - \Delta$ or larger than the value at index $z(u) + \Delta$.*

Per Branch call, ConTree keeps track of the set of threshold indices that may yield a split that improves the current best tree. This set is represented as a set of index intervals Q. Initially, Q contains one interval of all indices: $Q = \{[1..m]\}$. After each Split call, ConTree updates the set Q by using three pruning functions $\mathcal{P}$ that return a list of pruned intervals from the current interval $[i..j]$, the current $\mathcal{UB}$, and one or more subproblem solutions θ. Next, we explain these three pruning functions: neighborhood pruning, interval shrinking, and sub-interval pruning.

Neighborhood pruning (NB)

After the misclassification score θ_u for a split point $u \in [i..j]$ is computed, neighborhood pruning uses the SLB to remove similar split points from consideration. A simplified illustration of this pruning technique can be seen in Fig. 8.1a. Using Cor. 8.3.2, we define two functions $\underline{A}$ and $\bar{A}$ that return the closest thresholds from u that could still improve on $\mathcal{UB}$.

$$\underline{A}(u, \Delta) = \max \left\{ u' \in [m] \mid U^f_{u'} < \mathcal{D}^f_{z(u)-\Delta} \right\}, \tag{8.7}$$

$$\bar{A}(u, \Delta) = \min \left\{ u' \in [m] \mid U^f_{u'+1} > \mathcal{D}^f_{z(u)+\Delta} \right\}. \tag{8.8}$$

The functions $\underline{A}$ and $\bar{A}$ can be implemented using binary search with time complexity $\mathcal{O}(\log(m))$. Using these, we define $\mathcal{P}_{\text{NB}}$ which yields two new intervals:

$$\mathcal{P}_{\text{NB}}([i..j], u, \Delta) = \{[i..\underline{A}(u, \Delta)], [\bar{A}(u, \Delta)..j]\}. \tag{8.9}$$

Example. Consider a continuous feature vector with values [0.4, 0.5, 0.5, 0.7, 0.8, 0.10] with split points [0.45, 0.6, 0.75, 0.9], where we have computed an optimal tree for the split on $\tau = 0.75$ with two more misclassifications than the current best solution $\mathcal{UB}$, that is, $\Delta = 2$. Therefore, any possibly improving split needs to move at least two instances from the left to the right (or vice versa), which means that only the split point $\tau = 0.45$ can yield an improving solution.

Interval shrinking (IS)

Interval shrinking is an extension of neighborhood pruning and acts whenever $\mathcal{UB}$ is updated. Given an interval $[i..j] \in Q$, IS searches for the largest threshold index $u \in \mathcal{V}$ smaller than i and the smallest threshold index $v \in \mathcal{V}$ larger than j, with $\mathcal{V}$ the set of split indices for which the misclassification score θ_u and θ_v are already computed. Using Cor. 8.3.2, IS then prunes the interval $[i..j]$ as illustrated in Fig. 8.1b. To search for these indices u and v, we define the function B that uses binary search over $\mathcal{V}$ with time complexity $\mathcal{O}(\log(m))$:

$$B([i..j], \mathcal{V}) = \left\langle \max_{u \in \mathcal{V}: u < i} u, \min_{v \in \mathcal{V}: v > j} v \right\rangle . \tag{8.10}$$

Additionally, IS uses the following theorem:

Theorem 8.3.3. *Let w by any split point with a solution θ_w with a left subtree misclassification score $\theta_{w,L}$ of zero. Then any split point $u < w$ will yield $\theta_u \geq \theta_w$. Similarly, if the right subtree misclassification score $\theta_{w,R}$ is zero, then for all $v > w$ also $\theta_v \geq \theta_w$.*

Proof. W.l.o.g., consider the left case. Since $u < w$, it holds that $\mathcal{D}(f \leq S_u^f) \subset \mathcal{D}(f \leq S_w^f)$ and $\mathcal{D}(f > S_u^f) \supset \mathcal{D}(f > S_w^f)$. Eq. (8.3) then implies that $\theta_{u,L} \leq \theta_{w,L} = 0$ and $\theta_{u,R} \geq \theta_{w,R}$. Thus $\theta_u = \theta_{u,R}$, $\theta_w = \theta_{w,R}$, and $\theta_u \geq \theta_w$. □

Let M_L and M_R represent the right-most (left-most) index with a zero left (right) misclassification score found so far, incremented (decremented) by one. By combining Cor. 8.3.2 and Theorem 8.3.3, we define:

$$\mathcal{P}_{\text{IS}}([i..j], u, v, \Delta_u, \Delta_v, M_L, M_R) = \\ [\max(i, M_L, \bar{A}(u, \Delta_u)), \min(j, M_R, \underline{A}(v, \Delta_v))] , \tag{8.11}$$

where $\Delta_u = \theta_u - \mathcal{UB}$ and $\Delta_v = \theta_v - \mathcal{UB}$.

Sub-interval pruning (SP)

Sub-interval pruning can prune an entire interval $[i..j]$ using the following theorem:

Theorem 8.3.4. *Let $[i..j]$ be any threshold index interval. Let θ_u and θ_v be optimal solutions for previously computed split points $u < i$ and $v > j$, with the corresponding left and right misclassification scores $\theta_{u,L}$, $\theta_{u,R}$, $\theta_{v,L}$, and $\theta_{v,R}$. Then, if $\theta_{u,L} + \theta_{v,R} > \mathcal{UB}$, any split point $w \in [i..j]$ cannot improve on $\mathcal{UB}$.*

Proof. $w \geq i > u$ and $w \leq j < v$ and thus $\theta_{w,L} \geq \theta_{u,L}$ and $\theta_{w,R} \geq \theta_{v,R}$. Therefore, $\theta_w = \theta_{w,L} + \theta_{w,R} > \mathcal{UB}$. □

From Theorem 8.3.4, we define:

$$\mathcal{P}_{SP}([i..j], \mathcal{UB}, \theta_{u,L}, \theta_{v,R}) = \begin{cases} \emptyset, & \text{if } \theta_{u,L} + \theta_{v,R} > \mathcal{UB} \wedge u < i \wedge v > j \\ [i..j], & \text{otherwise} . \end{cases} \tag{8.12}$$

Algorithm 11: Branch($\mathcal{D}, d, f, \mathcal{UB}$) uses the pruning techniques to find the optimal threshold for feature f.

$\theta_{\text{opt}} \leftarrow \min_{\hat{y} \in \mathcal{Y}} \sum_{(x,y) \in \mathcal{D}} \mathbb{1}(\hat{y} \neq y)$
$M_L \leftarrow 0, M_R \leftarrow m + 1$
$Q \leftarrow \{[1..m]\}, \mathcal{V} \leftarrow \emptyset$
while $|Q| > 0$ **do**
  $[i..j] \leftarrow Q.\text{pop}()$
  $u, v \leftarrow B([i..j], \mathcal{V})$
  $\Delta_u \leftarrow \theta_u - \mathcal{UB}, \Delta_v \leftarrow \theta_v - \mathcal{UB}$
  $[i..j] \leftarrow \mathcal{P}_{\text{IS}}([i..j], u, v, \Delta_u, \Delta_v, M_L, M_R)$
  $[i..j] \leftarrow \mathcal{P}_{\text{SP}}([i..j], \mathcal{UB}, \theta_{u,L}, \theta_{v,R})$
  if $|[i..j]| = 0$ **then continue**
  $w \leftarrow \lfloor \frac{i+j}{2} \rfloor$
  if $d = 2$ **then** $\theta_{w,L}, \theta_{w,R} \leftarrow \text{D2Split}(\mathcal{D}, f, w)$
  else
    $\mathcal{D}_L \leftarrow \mathcal{D}(f \leq S_w^f), \mathcal{D}_R \leftarrow \mathcal{D}(f > S_w^f)$
    $\theta_{w,L} \leftarrow \text{CT}(\mathcal{D}_L, d-1, \mathcal{UB})$
    $\eta \leftarrow \min(z(w) - z(i), z(j) - z(w))$
    $\mathcal{UB}_R \leftarrow \max(\mathcal{UB} - \theta_{w,L}, \eta)$
    if $\mathcal{UB}_R \leq 0$ **then** $\theta_{w,R} \leftarrow \theta_{\text{opt}} - \theta_{w,L}$
    else $\theta_{w,R} \leftarrow \text{CT}(\mathcal{D}_R, d-1, \mathcal{UB}_R)$
  $\theta_w \leftarrow \theta_{w,L} + \theta_{w,R}$
  if $\theta_{w,L} = 0$ **then** $M_L \leftarrow w + 1$
  if $\theta_{w,R} = 0 \wedge \mathcal{UB}_R > 0$ **then** $M_R \leftarrow w - 1$
  if $\theta_w < \theta_{\text{opt}}$ **then**
    $\mathcal{UB} \leftarrow \min(\mathcal{UB}, \theta_w), \theta_{\text{opt}} \leftarrow \theta_w$
    if $\theta_{\text{opt}} = 0$ **then break**
  $Q.\text{push}\,(\mathcal{P}_{\text{NB}}([i..j], w, \max(1, \theta_w - \mathcal{UB})))$
  $\mathcal{V} \leftarrow \mathcal{V} \cup \{w\}$
return θ_{opt}

8.3.2. General Recursive Case

To eliminate the exploration of unnecessary splits in the recursion of Eq. (8.4), we use the pruning mechanisms presented above and keep track of upper bounds.

Interval pruning In each Branch call, described in Eq. (8.5), for each feature, we keep track of a set of intervals Q that may still contain the optimal split. We then choose a split point from one of these intervals, compute an optimal solution for this split, and then prune the search space according to the techniques mentioned above. When Q is empty, we have arrived at the optimal solution.

The pseudo-code of this procedure is shown in Alg. 11. It loops over the set of intervals $[i..j] \in Q$. Using Eq. (8.10), it finds $u, v \in \mathcal{V}$, the closest indices outside of $[i..j]$ for which the misclassification score is already computed. First, it attempts

Algorithm 12: D2Split$(\mathcal{D}, f_1, w)$ finds splits of depth two more efficiently than the normal recursion.

$\theta_L \leftarrow |\mathcal{D}|, \theta_R \leftarrow |\mathcal{D}|$
$FQ_L^y \leftarrow 0 \quad \forall y \in \mathcal{Y}$
for $(x, y) \in \mathcal{D}(f \leq S_w^{f_1})$ **do** $FQ_L^y \leftarrow FQ_L^y + 1$
$FQ_R^y \leftarrow FQ^y - FQ_L^y \quad \forall y \in \mathcal{Y}$
for $f_2 \in \mathcal{F}$ **do**
  $C_L^y \leftarrow 0, C_R^y \leftarrow 0 \quad \forall y \in \mathcal{Y}$
  for $(x, y) \in \mathcal{D}$ *sorted by* f_2 **do**
    if $x_{f_1} \leq S_w^{f_1}$ **then**
      $\theta_{LL} \leftarrow \min_{\hat{y}}(C_L^{\hat{y}})$
      $\theta_{LR} \leftarrow \min_{\hat{y}}(FQ_L^{\hat{y}} - C_L^{\hat{y}})$
      if $\theta_{LL} + \theta_{LR} \leq \theta_L$ **then**
        $\theta_L \leftarrow \theta_{LL} + \theta_{LR}$
      $C_L^y \leftarrow C_L^y + 1$
    else
      Same logic for instances going right
    if $\theta_L + \theta_R = 0$ **then break**
return θ_L, θ_R

to reduce the interval using SP and IS. Then we need to select any point $w \in [i..j]$. If w is close to previously computed splits, only a little new information is gained. Therefore, we choose the midpoint $w = \lfloor \frac{i+j}{2} \rfloor$.

Then, if the remaining tree depth budget is two, we use a special subroutine which is explained in the subsection below. Otherwise, the dataset is split and a recursive CT call is made to get the optimal left subtree. This left solution is used to compute an upper bound for the right subtree. Only if the right upper bound indicates that an improving solution can still be found, is another recursive call made to get the right optimal subtree. If either the left or right subtree has zero misclassifications, the M_L or M_R indices are updated accordingly. If a better solution is found, the upper bound is updated. If a tree with zero misclassifications is found, the search is done. Finally, the remaining interval is divided into two using NB, both of which are added to Q, and the search continues with the next interval in Q.

Upper bounds If we set the upper bound for the right subtree tightly to the bound of what right solution can improve the current solution ($\mathcal{UB}_R \leftarrow \mathcal{UB} - \theta_{w,L}$), the right subproblem can be terminated early if no such solution exists. However, when doing so, we do not gain any information for pruning and we observed that this therefore typically decreases performance. On the other hand, no upper-bound-based pruning at all results in many unnecessary recursive calls. In this trade-off, we settled on a hybrid approach that in practice works well: the right upper bound $\mathcal{UB}_R$ is set to the maximum of $\mathcal{UB} - \theta_{w,L}$ and the length η of the longest side of the interval edges $z(i)$ and $z(j)$ to the midpoint $z(w)$.

8.3.3. Depth-Two Subroutine

To improve runtime, Demirović et al. (2022) introduced a specialized subroutine for trees of depth-two that is more efficient than doing recursive calls. However, it requires a quadratic amount of memory in terms of the number of binary features, which is problematic if we consider a binary feature for every possible threshold on continuous data.

Instead, we provide a specialized subroutine that simultaneously finds an optimal left and right subtree of a depth-two split and does not have this quadratic memory consumption by exploiting the fact that we can sort the observations by their feature values. Since splitting the data preserves the order, we sort the dataset once in the beginning. Then for the sorted data, Alg. 12 shows how an optimal depth-two tree can be found in $\mathcal{O}(|\mathcal{D}||\mathcal{F}|)$ for a given split point w on feature f_1 for the root node of the subtree. The core idea is that we first count with the variables FQ_L^y and FQ_R^y how many observations of class y go to the left and right by splitting at point w. Then, when deciding on the second splitting feature f_2 (of either the left or right subtree), we traverse all observations sorted by f_2 and incrementally keep track of the current counts per label C_L^y and C_R^y. Based on these two label counts, the label counts for all splitting thresholds on f_2 for all four leaf nodes of a depth-two tree can be determined as the dataset is traversed:

$$\begin{aligned} C_{LL}^y &= C_L^y\,, & C_{LR}^y &= FQ_L^y - C_L^y\,, \\ C_{RL}^y &= C_R^y\,, & C_{RR}^y &= FQ_R^y - C_R^y\,. \end{aligned} \tag{8.13}$$

Alg. 12 shows how the minimal misclassifications in the subtrees θ_{LL} and θ_{RR} can be computed based on these values.

8.3.4. Optimality Gap

We add a max-gap parameter that determines how far from optimal the final solution is allowed to be. This increases the pruning strength at the expense of optimality. For example, for NB, the distance Δ to a possibly improving split is now computed as $\Delta = \max(1, \theta_u - (\mathcal{UB} - \text{max-gap}))$. To use the gap parameter across multiple depths of the search, we set the max-gap for the current depth to half of the total. The other half is distributed evenly over the two subproblems. This allows a trade-off between training time and accuracy.

8.3.5. Comparison to Previous Bounding Methods

Mazumder et al. (2022) propose three lower bounds that require splitting the data into quantiles for a given feature. Their first lower bound is similar to our sub-interval pruning, but our definition is independent of the remaining depth budget. Their other lower bounds are tighter, but also more expensive to compute. Future work could investigate using such or similar lower bounds in ConTree.

The similarity-based lower bound (SLB) was proposed in previous work (Hu et al., 2019; Lin et al., 2020; Demirović et al., 2022), but our application of the bound is more efficient. Lin et al. (2020) point out that the *similar support* bound in OSDT (Hu et al., 2019) is too expensive to compute frequently. Therefore, they

Dataset	$\|\mathcal{D}\|$	$\|\mathcal{F}\|$	$\|\mathcal{Y}\|$	OCT	RS-OCT	SAT-Shati	Quant-BnB	ConTree No D2	ConTree D2
Avila	10430	10	12	(100%)	(46%)	> 4h	2	63	**0.1**
Bank	1097	4	2	(31%)	177	62	0.5	0.1	**< 0.1**
Bean	10888	16	7	(100%)	(24%)	> 4h	4	199	**0.2**
Bidding	5056	9	2	(100%)	(67%)	672	0.9	1	**0.1**
Eeg	11984	14	2	OoM	(165%)	> 4h	4	178	**0.2**
Fault	1552	27	7	(100%)	(126%)	> 4h	2	240	**0.1**
Htru	14318	8	2	OoM	(301%)	> 4h	2	929	**0.2**
Magic	15216	10	2	(100%)	(88%)	> 4h	2	42	**0.3**
Occupancy	8143	5	2	(100%)	(9%)	355	0.9	2	**< 0.1**
Page	4378	10	5	(100%)	(136%)	2836	1	3	**< 0.1**
Raisin	720	7	2	(99%)	(15%)	485	0.7	1	**< 0.1**
Rice	3048	7	2	(100%)	(54%)	> 4h	1	24	**0.1**
Room	8103	16	4	(100%)	(88%)	4327	2	10	**< 0.1**
Segment	1848	18	7	(100%)	2896	442	2	54	**0.1**
Skin	196045	3	3	-	(43%)	> 4h	3	63	**0.1**
Wilt	4339	5	5	(100%)	(35%)	68	0.9	2	**< 0.1**

Table 8.1: Runtime (s) for optimizing depth-two trees of OCT, RS-OCT, SAT-Shati, Quant-BnB, and ConTree (with and without the depth-two subroutine). Runtimes are averaged over twenty runs. Values in parentheses show the optimality gap at time out. OoM means out of memory. '-' means the linear relaxation was unsolved at the time limit. Best results are in bold.

propose to use *hash trees* to identify similar subtrees but provide no further details. The implementation of their method, GOSDT, computes the difference between two subproblems by computing the xor of bit-vectors that represent the dataset corresponding to the subproblems. Demirović et al. (2022) loop once over the two sorted lists of identifiers of the datasets to count the differences. Both of these approaches require $\mathcal{O}(n)$ operations to compute the difference. ConTree, on the other hand, exploits the properties of sorted numeric feature data and computes the bound in $\mathcal{O}(1)$ by computing the difference between the two split indices. It applies the bound using the novel pruning techniques in $\mathcal{O}(\log(m))$.

8.4. Experiments

Our experiments aim to answer the following questions: 1) what is the effect of the pruning techniques and the depth-two subroutine; 2) how does ConTree's runtime compare to state-of-the-art ODT algorithms; and 3) what is ConTree's anytime performance? In the appendices 8.B, 8.C, and 8.D, we additionally answer the following questions: 4) how does ConTree's memory usage compare to previous methods; 5) how well does ConTree scale to larger depth limits; and 6) how does ConTree's out-of-sample performance compare to CART and ODTs on binarized data?

The results show that our pruning techniques and depth-two subroutine make ConTree one or more orders of magnitude faster than the state-of-the-art optimal

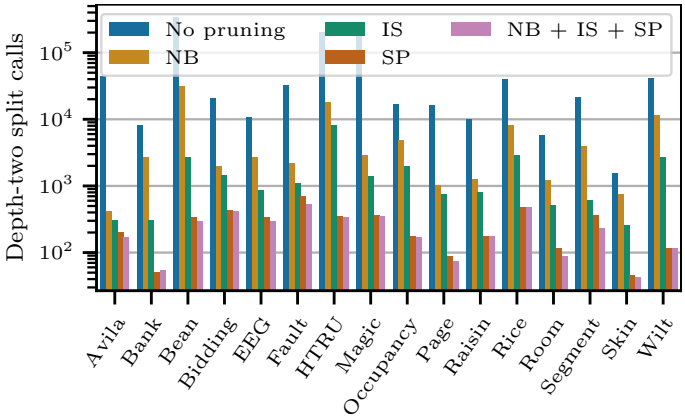


Figure 8.2: The number of D2Split calls for no pruning, the three pruning techniques, and all three combined.

methods while using significantly less memory. Additionally, its out-of-sample performance is better than both CART and ODTs on binarized data. Finally, good solutions are often found early but much time is spent proving optimality.

Setup We have implemented ConTree in C++ and provide it as a Python package.[1] All computations were performed on an Intel Xeon E5-6448Y 32C 2.1GHz processor with 25GB RAM running Linux Red Hat Enterprise 8.10. We set a timeout of four hours. We evaluate on 16 datasets from the UCI repository (Dua and Graff, 2017, see Appendix 8.A). Unless specified otherwise, we run ConTree with its max-gap set to zero, thus yielding optimal solutions.

Pruning techniques Fig. 8.2 shows the impact of the neighborhood pruning (NB), interval shrinking (IS), and sub-interval pruning (SP) techniques in comparison to the baseline with no pruning for trees of depth two. On average, all pruning techniques significantly prune the search space, with NB, IS, and SP respectively pruning on average 91.1%, 97.5%, and 99.6% of all D2Split calls, while retaining the optimal solution. SP generally provides the best results since it efficiently prunes entire intervals. Combining all three methods (NB+IS+SP) on average yields a 10% reduction of D2Split calls in comparison to only using SP.

Depth-two subroutine The last two columns of Table 8.1 show the performance of ConTree with the depth-two subroutine D2Split versus without (ConTree with "No D2") for computing depth-two trees. Both methods use all the pruning techniques. Averaged over twenty runs, the depth-two solver improves the computation time by a factor of 320 compared to the baseline (geometric mean).

Runtime Mazumder et al. (2022) compared Quant-BnB with BinOCT (Verwer and Zhang, 2019), DL8.5 (Aglin et al., 2020a), and MurTree (Demirović et al., 2022), each of which requires explicit binarization. Quant-BnB scales one or more orders

[1] `https://github.com/ConSol-Lab/contree`.

	Depth = 3		Depth = 4	
Dataset	Quant-BnB	ConTree	ConTree	ConTree with $\leq 1\%$ Gap
Avila	4451	24	4195	3237
Bank	2	< 0.1	< 0.1	< 0.1
Bean	583	61	$>$ 4h	3640
Bidding	15	2	5	< 0.1
Eeg	8535	136	$>$ 4h	$>$ 4h
Fault	$>$ 4h	55	12331	8592
Htru	13147	74	$>$ 4h	191
Magic	1419	60	$>$ 4h	5719
Occupancy	76	0.4	17	0.1
Page	388	2	499	63
Raisin	65	0.5	65	27
Rice	817	12	2215	231
Room	92	1	44	0.1
Segment	64	2	191	75
Skin	218	10	211	5
Wilt	26	0.1	0.3	< 0.1

Table 8.2: Runtime (s) comparison between Quant-BnB and ConTree. Averaged over twenty runs. For depth three, ConTree is on average one or two orders of magnitude faster than Quant-BnB. ConTree's runtime can be significantly reduced by setting a permissible optimality gap.

of magnitude better than all of these methods when those are trained with binary features for every possible threshold. Additionally, they report that the optimal methods OCT (Bertsimas and Dunn, 2017), GOSDT (Lin et al., 2020), FlowOCT, and BendersOCT (Aghaei et al., 2024) cannot solve any of their datasets within a four-hour time limit.

We compare ConTree to the MIP methods OCT and RS-OCT (Hua et al., 2022), the SAT method by Shati et al. (2023b), and Quant-BnB (Mazumder et al., 2022).[2] Each of these methods optimizes ODTs directly on the numeric feature data. We initialize both OCT and RS-OCT with a warm start from CART and set the node cost to zero. Both OCT and RS-OCT could use up to eight threads. OCT is solved with Gurobi 9.5.2. Each method is run twenty times on each dataset, except when it exceeds the four-hour time-out. For the MIP methods, we report the optimality gap at time-out.

Table 8.1 shows the runtime results for optimizing trees of depth two. Even after four hours, the MIP methods typically have a large optimality gap remaining. OCT ran out of memory twice and once did not solve the linear relaxation before the time out. The SAT approach performs better but also hits the time-out for seven datasets. Quant-BnB and ConTree, on the other hand, run in the (sub-)second range. Therefore, we further compare Quant-BnB and ConTree.

Table 8.2 shows that for depth three, ConTree outperforms Quant-BnB on

[2] `https://github.com/LucasBoTang/Optimal_Classification_Trees`, `https://github.com/YankaiGroup/optimal_decision_tree`, `https://github.com/HarisRasul12/ESC499-Thesis-SAT-Trees`, `https://github.com/mengxianglgal/Quant-BnB`.

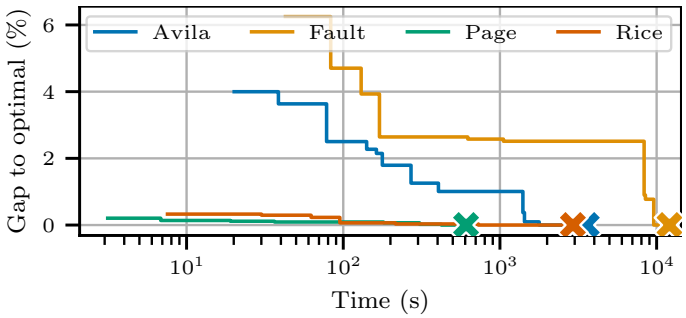


Figure 8.3: The distance to the optimal solution for ConTree's best depth-four solution over time for three datasets. The optimal solution is typically found significantly earlier than the end of the search (the final cross).

average by a factor 63 (geometric mean), ranging from 7 times faster for Bidding up to 266 times faster for Fault. The comparisons of Quant-BnB with BinOCT, DL8.5, MurTree, GOSDT, OCT, FlowOCT, and BendersOCT by Mazumder et al. (2022) combined with our new runtime results show that ConTree outperforms state-of-the-art optimal methods by one or more orders of magnitude.

Quant-BnB's implementation only permits training trees up to depth three, so for depth four, we report only ConTree's performance. For all but four datasets, ConTree finds the optimal depth-four tree within the time limit. Appendix 8.B further shows that for depth five and six, ConTree finds and proves optimal solutions within the time limit for eight and seven datasets respectively.

The last column in Table 8.2 shows the runtime ConTree requires to find a depth-four tree that is provably within 1% of the optimal solution (with max-gap = $\lfloor 0.01|\mathcal{D}| \rfloor$). This significantly reduces ConTree's runtime and allows a user to make a trade-off between training accuracy and runtime.

Anytime performance Fig. 8.3 shows the best solution found by ConTree at any time during a depth-four search, expressed as the distance to the optimal solution for the Avila, Fault, Page, and Rice datasets. For the Fault dataset, a final improving solution is found late after the start of the search. In the other three cases, a good solution is found early and most of the time is spent on proving optimality.

8.5. Conclusion

We introduce ConTree, a dynamic programming and branch-and-bound algorithm that outperforms state-of-the-art algorithms for training optimal classification trees with continuous features by one or more orders of magnitude. This result is obtained through three novel pruning techniques that on average prune 99.6% of recursive calls, and a depth-two subroutine that computes splits 320 times faster than a naive approach. Moreover, ConTree obtains an out-of-sample accuracy for depth-three trees 5% higher than CART and 0.7% higher than optimal decision trees trained with a coarse binarization. These results enable the application of optimal decision

trees for real-world scenarios.

Future work could add a node cost or node limit to further encourage sparsity and extend ConTree to regression by using bounds from Zhang et al. (2023) and Van den Bos et al. (2024). Finally, branching on the power set of categorical variables as done by Shati et al. (2023b), could be explored.

Appendices for Chapter 8

8.A. Data Preprocessing

All datasets used in the experiments are from the UCI machine learning repository Dua and Graff, 2017. We apply the same data preprocessing techniques as Mazumder et al. (2022): all features that do not assist prediction are removed (e.g., timestamps or unique identifiers). For the experiments that measure the runtime and memory usage, we use the train split from their repository.[3] For the out-of-sample experiments in Appendix 8.D, we use the full dataset. The preprocessing per dataset is as follows:

Avila (Stefano et al., 2018) All features are kept as is.

Bank (Lohweg, 2013) We keep all the features from the *Banknote Authentication* dataset.

Bean (Koklu and Ozkan, 2020) We keep all the features from the *Dry Bean* dataset.

Bidding (*Shill Bidding Dataset* 2020) We remove the features *Record_ID*, *Auction_ID*, and *Bidder_ID* from the *Shill Bidding* dataset.

Eeg (Roesler, 2013) We keep all the features from the *EEG Eye State* dataset.

Fault (Buscema et al., 2010) We keep all the features from the *Steel Plates Faults* dataset.

Htru (Lyon et al., 2016) We keep all the features from the *HTRU2* dataset.

Magic (Bock, 2007) We keep all the features from the *MAGIC Gamma Telescope* dataset.

Occupancy (Candanedo and Feldheim, 2016) We remove the *id* and *date* features from the *Occupancy Detection* dataset.

Page (Malerba, 1995) We keep all the features from the *Page Blocks Classification* dataset.

Raisin (Çınar et al., 2020) All features are kept as is.

Rice (Çınar and Koklu, 2019) We keep all the features from the *Rice (Cammeo and Osmancik)* dataset.

Room (Singh et al., 2018) We remove the *Date* and *Time* features from the *Room Occupancy Estimation* dataset.

Segment (*Image Segmentation Dataset* 1990) We remove the *REGION-PIXEL-COUNT* feature from the *Image Segmentation* dataset because it has only one unique value.

Skin (Bhatt and Dhall, 2012) We keep all the features from the *Skin Segmentation* dataset.

Wilt (Johnson, 2014) All features are kept as is.

[3] `https://github.com/mengxianglgal/Quant-BnB`

	Depth = 5, Optimal			Depth = 5, ≤ 1% Gap		
Dataset	Accuracy (%)	Runtime (s)	Memory (MB)	Accuracy (%)	Runtime (s)	Memory (MB)
Avila	**67.9**	> 4h	44	**67.9**	> 4h	41
Bank	**100**	< 0.1	4	99.9	< 0.1	4
Bean	**92.4**	> 4h	34	**92.4**	> 4h	33
Bidding	**100**	10	11	99.9	< 0.1	9
Eeg	**75.5**	> 4h	30	**75.5**	> 4h	29
Fault	**76.4**	> 4h	12	74.0	> 4h	12
Htru	98.4	> 4h	37	**98.5**	> 4h	61
Magic	81.3	> 4h	34	**86.6**	> 4h	40
Occupancy	**99.9**	119	39	99.7	0.3	10
Page	**98.6**	> 4h	161	**98.6**	3647	55
Raisin	**95.4**	1199	50	95.3	489	23
Rice	**95.6**	> 4h	74	95.5	> 4h	84
Room	**100**	7	21	99.9	0.2	18
Segment	**99.1**	3715	51	99.0	776	20
Skin	**99.8**	1705	782	99.6	13	139
Wilt	**100**	< 0.1	7	99.9	< 0.1	7

Table 8.3: ConTree results for maximum depth five. Runtime is averaged over five runs. The maximum memory usage observed is stated. If ConTree exceeds the time limit we report the best solution found at time-out. Best accuracy results are put in bold.

To determine all unique values per feature in a dataset, we sort the instances and check if consecutive values differ more than a small value ε. In this chapter, we set $\varepsilon = 1 \times 10^{-7}$.

8.B. Results for Larger Depth Limits

Table 8.3 and 8.4 show the results for training optimal decision trees of depths five and six. The results are averaged over five runs. ConTree finds and proves optimal trees for eight and seven datasets respectively.

These tables also show that when an optimality gap of 1% is allowed, the runtime is sometimes significantly reduced. There are also cases where the final training accuracy is higher when running with this optimality gap. This is because the permitted gap allows more of the search space to be pruned and less time is spent in checking every possible split. This motivates future work in improving the anytime performance of ConTree and in establishing an improved search order.

8.C. Memory Usage Results

Table 8.5 reports the memory usage of the SAT-Shati, Quant-BnB, and ConTree for computing trees of depth two (all methods), three (Quant-BnB and ConTree), and four (only ConTree). For depth two, memory poses no problem for all three methods. For depth three, Quant-BnB uses 15GB of memory for Eeg and Htru and even 25GB

	Depth = 6, Optimal			Depth = 6, ≤ 1% Gap		
Dataset	Accuracy (%)	Runtime (s)	Memory (MB)	Accuracy (%)	Runtime (s)	Memory (MB)
Avila	**70.5**	> 4h	52	**70.5**	> 4h	51
Bank	**100**	< 0.1	5	**100**	< 0.1	5
Bean	**73.7**	> 4h	56	**73.7**	> 4h	64
Bidding	**100**	< 0.1	10	99.9	< 0.1	9
Eeg	**73.5**	> 4h	33	**73.5**	> 4h	34
Fault	**78.1**	> 4h	15	78.0	> 4h	15
Htru	**98.6**	> 4h	57	98.5	> 4h	72
Magic	**74.4**	> 4h	45	74.3	> 4h	47
Occupancy	**100**	34	33	99.8	0.5	11
Page	98.0	> 4h	161	**98.3**	> 4h	169
Raisin	**99.6**	180	60	99.4	8	7
Rice	**96.8**	> 4h	320	96.7	> 4h	201
Room	**100**	2	20	**100**	0.2	20
Segment	**100**	483	14	99.8	154	10
Skin	**99.9**	> 4h	20105	99.8	22	176
Wilt	**100**	< 0.1	7	99.9	< 0.1	7

Table 8.4: ConTree results for maximum depth six. Runtime is averaged over five runs. The maximum memory usage observed is stated. If ConTree exceeds the time limit we report the best solution found at time-out. Best accuracy results are put in bold.

for Fault. In contrast, ConTree's memory usage for these datasets is 18MB, 15MB, and 7MB respectively. At depth four, ConTree's maximum memory usage is for the Skin dataset, where it uses 123MB of memory. This highlights ConTree's memory efficiency in comparison to previous methods.

Table 8.3 and Table 8.4 further show the maximum memory usage when computing optimal trees of depths five and six. For all datasets, except Skin, the memory consumption stays within 320MB. The Skin dataset is the only dataset where ConTree uses a lot of memory when optimizing deeper trees. The reason for this is the size of the dataset which is ten times larger than the next largest dataset in our benchmark. If memory becomes an issue, ConTree's caching mechanism could be changed from dataset caching to branch caching to lower memory consumption (Demirović et al., 2022).

8.D. Out-of-Sample Results

To test the out-of-sample performance of ConTree, we compare it to the CART heuristic, and to optimal decision trees trained on binarized data. We test two binarization approaches: one based on *quantiles* and the other based on *threshold guessing* using a reference ensemble (McTavish et al., 2022). For the quantile approach, we binarize each of the numeric features using thresholds on ten quantiles of the feature distribution. For the threshold guessing approach, we follow McTavish et al. (2022) and train a gradient boosting classifier with the number of estimators

	Depth = 2			Depth = 3		Depth = 4
Dataset	SAT-Shati	Quant-BnB	ConTree	Quant-BnB	ConTree	ConTree
Avila	944	517	11	4319	13	20
Bank	208	477	4	516	4	4
Bean	934	532	15	1220	19	26
Bidding	446	495	7	579	8	10
Eeg	1466	509	14	14850	18	25
Fault	754	490	6	24750	7	10
Htru	1256	518	12	14781	15	24
Magic	1530	533	15	1458	18	26
Occupancy	450	494	7	608	8	11
Page	490	493	7	1184	8	11
Raisin	283	489	4	550	4	5
Rice	816	494	5	1198	6	9
Room	693	504	12	646	14	18
Segment	306	503	6	595	6	8
Skin	3965	571	78	736	89	123
Wilt	254	494	6	564	6	7

Table 8.5: Maximum memory usage (MB) per method. ConTree uses significantly less memory than previous approaches.

set to ten times the number of continuous features and the maximum depth to two. After training, the least important features are removed iteratively until the performance of the ensemble drops.[4]

We repeat the experiment five times on five stratified 80%-20% train-test splits. For this, we use the full datasets rather than only the training splits used in previous experiments. In Table 8.6 we list the size of these datasets, the number of binary features $|\mathcal{F}_b|$ obtained by the two binarization methods, and the number of binary features one would need to search over all possible thresholds on the numeric features (what ConTree does, but without an explicit binarization).

Since all optimal methods obtain trees with the same accuracy, we only need to compare to one method on the binarized data. Because of its scalability, we train optimal decision trees for the binarized data using STreeD (Van der Linden et al., 2023).[5] For CART we use the sklearn implementation. Each method is trained with a maximum depth of three. Within that depth limit, each method tunes a tree-size hyperparameter using five 80%-20% train-validation splits. CART uses cost-complexity tuning, STreeD tunes the depth and the number of nodes, and ConTree only tunes the depth.

Table 8.6 shows how ConTree significantly outperforms both CART on the continuous data and optimal decision trees on binarized data. ConTree's test accuracy on average is 4.7% higher than CART. For the Segment dataset, this difference is even 30%. ConTree's test accuracy on average is 0.7% higher than the

[4] `https://github.com/ubc-systopia/gosdt-guesses`.
[5] `https://github.com/AlgTUDelft/pystreed`.

			Optimal decision tree (given the binarization)					
		CART	10 quantiles		Guessing		ConTree (All τ)	
Dataset	$\lvert\mathcal{D}\rvert$	Acc.	$\lvert\mathcal{F}_b\rvert$	Acc.	$\lvert\mathcal{F}_b\rvert$	Acc.	$\lvert\mathcal{F}_b\rvert$	Acc.
Avila	20867	52.7±0.1	95	56.6±0.1	108	56.8±0.2	41110	**58.3±0.1**
Bank	1372	92.7±0.9	40	96.5±0.3	29	96.0±0.3	5016	**97.0±0.3**
Bean	13611	77.6±0.1	160	84.6±0.2	194	79.9±0.5	211343	**86.8±0.1**
Bidding	6321	98.4±0.1	61	**99.3±0.1**	23	**99.3±0.1**	12527	**99.3±0.1**
Eeg	14980	66.3±0.2	140	69.7±0.2	214	**70.7±0.2**	5404	70.6±0.3
Fault	1941	52.7±0.5	236	65.1±0.6	252	65.4±1.1	19226	**65.6±0.9**
Htru	17898	97.7±0.1	80	97.7±0.1	99	**97.8±0.1**	123368	**97.8±0.1**
Magic	19020	79.0±0.3	100	82.1±0.2	198	**82.5±0.3**	146815	82.4±0.2
Occupancy	20560	98.9±0.1	45	98.3±0.2	31	98.9±0.1	19709	**99.0±0.1**
Page	5473	96.0±0.2	98	94.8±0.3	87	96.0±0.1	9082	**96.3±0.3**
Raisin	900	84.8±0.7	70	**85.7±1.2**	62	84.6±0.6	6289	85.2±0.7
Rice	3810	93.2±0.5	70	92.9±0.3	81	**93.5±0.5**	24635	93.2±0.5
Room	10129	96.9±0.1	96	98.1±0.2	67	98.8±0.1	3072	**99.0±0.1**
Segment	2310	56.8±0.1	165	85.2±0.6	34	81.8±2.3	12057	**87.1±0.8**
Skin	245057	96.5±0.0	30	96.1±0.0	36	96.7±0.0	765	**96.8±0.0**
Wilt	4839	97.5±0.2	50	97.7±0.2	61	97.8±0.2	22575	**97.9±0.1**
Wins		0		2		5		12

Table 8.6: Out-of-sample accuracy and standard error (%) for trees of maximum depth three for five runs. ConTree (being able to split on all thresholds) significantly outperforms CART on the numeric data and optimal decision trees on data binarized using ten thresholds per feature or using a threshold guessing method (McTavish et al., 2022). The number of binary features considered are mentioned in the columns marked with $|\mathcal{F}_b|$. For ConTree, the column $|\mathcal{F}_b|$ mentions the number of binary features that would be required for an explicit binarization.

trees optimized with the quantile binarization and 1.0% higher than trees optimized with the threshold guessing binarization. A Wilcoxon signed rank test indicates that all these results are statistically significant ($p < 0.01$).

9

Enumerating the Rashomon Set of Close-to-Optimal Trees

Preface In all previous chapters, we have optimized a single decision tree model. Using the framework from Chapter 4, this chapter provides scalability improvements for computing the complete set of all close-to-optimal models, i.e., the Rashomon set, and shows how this set can be computed anytime and in order, while generalizing to a broad variety of learning tasks. The Rashomon set can be used, for example, to improve variable importance analysis, and to optimize complex objectives that are hard to optimize directly.

Abstract Sparse decision tree learning provides accurate and interpretable predictive models that are ideal for high-stakes applications by finding the single most accurate tree within a (soft) size limit. Rather than relying on a single "best" tree, Rashomon sets—trees with similar performance but varying structures—can be used to enhance variable importance analysis, enrich explanations, and enable users to choose simpler trees or those that satisfy stakeholder preferences (e.g., fairness) without hard-coding such criteria into the objective function. However, because finding the optimal tree is NP-hard, enumerating the Rashomon set is inherently challenging. Therefore, we introduce SORTD, a novel framework that improves scalability and enumerates trees in the Rashomon set in order of the objective value, thus offering anytime behavior. Our experiments show that SORTD reduces runtime by up to two orders of magnitude compared with the state of the art. Moreover, SORTD can compute Rashomon sets for any separable and totally ordered objective and supports post-evaluating the set using other separable (and partially ordered) objectives. Together, these advances make exploring Rashomon sets more practical in real-world applications.

Parts of this chapter have been published in Arslan, Van der Linden, Hoogendoorn, Rinaldi, and Demirović (2025), "SORTeD Rashomon Sets of Sparse Decision Trees: Anytime Enumeration", in *Advances in NeurIPS-25*. This was joint work with Elif Arslan, a fellow PhD candidate, who did most of the implementation, investigation, visualization, and writing. I contributed most of the methodology, and helped in implementation, investigation, visualization, and writing.

9.1. Introduction

While decision tree methods have been extensively studied, most method return a *single* decision tree that is constructed to perform well with respect to a loss function. However, there are typically *many* approximately equally good—but possibly very different—decision-tree models for a given learning problem. This phenomenon is known as the *Rashomon effect* (Breiman, 2001), and the set of all models within a tolerance of the globally optimal loss value is called the *Rashomon set*.

Rashomon sets offer several advantages over relying on a single best model. In explainable AI, they enable the discovery of simpler models (Semenova et al., 2022; Rudin et al., 2024), support diverse counterfactual explanations (Andersen et al., 2023), address underspecification (D'Amour et al., 2022), also in explanations (Laberge et al., 2023), and improve variable-importance analysis (Donnelly et al., 2023). In addition, they allow evaluation of criteria that are difficult to encode directly into the objective function—such as fairness or robustness—by analysing properties across the set. In recidivism prediction, Fisher et al. (2019) examined the influence of bias-associated variables, and Marx et al. (2020) explored how similar models might yield conflicting predictions. Rashomon sets also provide a more flexible interface for stakeholders, enabling them to select models that best align with their domain-specific constraints. In healthcare, a diverse set of models obtained from the Rashomon set was used to support informed decision-making (Kobylińska et al., 2024; Zhu et al., 2025).

Benefiting from the Rashomon set requires efficient evaluation, as the set size can be prohibitively large (with only ten features and a depth budget of four, it may exceed 10^{12} trees). Practical use cases often require finding the most accurate trees, while also satisfying structural constraints (e.g., limits on tree size or mandatory/forbidden features) or attaining favourable scores on complementary objectives (e.g., fairness). This search task would be greatly simplified if candidate trees were generated in non-decreasing order of their objective value, preventing users from spending substantial time to examine the entire set to locate the desired models.

The current state of the art for obtaining the Rashomon set of sparse decision trees (Xin et al., 2022) returns every tree whose performance falls within a user-chosen tolerance of the best model (e.g., 5% above the optimum) but the output is not ordered by the objective. If the search is terminated early, e.g., because of a time-out, there is no guarantee that the returned set contains the actually best models according to the objective. Furthermore, when the appropriate margin is uncertain, it is often more practical to select a larger value, which can result in the construction of an extremely large set; conversely, if a fixed number of top-ranked models is of interest, there is no systematic mechanism to terminate enumeration once that quantity has been reached.

To address these concerns, we propose a novel framework, **SORTD** (**So**rted **R**ashomon Sets of **T**rees using **D**ynamic Programming). In doing so, we achieve three contributions. First, we compute the Rashomon set *in order*: trees with the best objective values are generated first. Ordered generation enables early termination when a specified number of high-quality models has been obtained, hence providing an anytime Rashomon set property. Our experiments show that

having access to the best models early can speed up downstream evaluation tasks such as variable-importance analysis.

Our second contribution is *improved scalability in the Rashomon set calculation*. To reduce runtime, SORTD incorporates a specialised algorithm for trees of depth two or less. This design allows SORTD to scale well with both the number of features and the depth budget. In our evaluation on a variety of benchmark classification datasets, we show that SORTD significantly outperforms the state-of-the-art, achieving speed-ups of up to *two orders of magnitude*.

Finally, our third contribution is *providing a general framework* that computes Rashomon sets for *separable and totally ordered objectives* and evaluates them with any separable and partially ordered one. We demonstrate this by enumerating the Rashomon set also for regression trees in addition to classification, and evaluating the Rashomon set of decision trees using an additional fairness objective. Together, these contributions make SORTD a practical and scalable tool for decision tree training, evaluation, and selection.

The rest of the chapter is organized as follows: Section 9.2 reviews related work; Section 9.3 covers preliminaries; Section 9.4 details the Rashomon set calculation; and Section 9.5 presents the experiments.

9.2. Related Work

Rashomon set computation has recently been explored for risk score models (Liu et al., 2022; Zhu et al., 2025), additive models (Semenova et al., 2022; Laberge et al., 2023; Zhong et al., 2023), rule sets (Hara and Ishihata, 2018; Ciaperoni et al., 2024), random forest (Laberge et al., 2023), and kernel ridge regression (Laberge et al., 2023). For sparse decision trees—the focus of this work—the only dedicated approach is TreeFARMS (Xin et al., 2022), which enumerates every tree within a given user-chosen tolerance (the "Rashomon multiplier") but does not preserve a global ordering of trees with respect to the objective. However, this user tolerance is rarely known a priori: an overly large value may result in generating billions of trees, which is memory and time intensive; while a small value may return too few trees. Additionally, TreeFARMS is tailored to classification, and extension to other optimization tasks is non-trivial.

On the contrary, our method produces solutions iteratively in non-decreasing order, allowing the algorithm to stop as soon as a target number of high-quality trees is reached without relying on an accurately tuned tolerance. This gives SORTD an anytime behavior: stopping the search at any time yields a Rashomon set. Furthermore, a specialised depth-two solver reduces runtime and improves scalability with both the number of features and the depth budget.

Additionally, SORTD handles any separable and totally ordered loss function and supports post-hoc evaluation of separable and partially ordered objectives (e.g., multi-objective optimization), hence increasing flexibility in learning and evaluation. For example, while Demirović and Stuckey (2021) develop a specialized DP algorithm for non-linear metrics such as F1-score, Xin et al. (2022) obtain the optimal F1-score tree from the Rashomon set based on optimizing accuracy. Similarly, rather than

building a custom method for each objective or constraint, such as for example, a demographic parity fairness constraint (Van der Linden et al., 2022), SORTD allows to explore the Rashomon set instead to find trees that are both accurate and fair.

9.3. Preliminaries

Notation Given a set of binary *features* F and a set of *labels* K, a sample (x_i, k_i) is a pair of feature vector $x_i \in \{0,1\}^{|F|}$ and label $k_i \in K$. A dataset $D = \{(x_i, k_i)\}_i$ is the set of samples that can be used to train a prediction model. Since we assume that the features are binary, we use $D(f)$ to indicate the set of samples where f is satisfied and $D(\bar{f})$ is the set of samples where f is not satisfied.

Sparse tree objective A binary tree is a function $T : \{0,1\}^{|F|} \to K$ that recursively maps a feature vector x to a predicted label $\hat{k}$. Starting with the root node, each internal node in T sends an instance left or right when its specified binary feature test is satisfied in x or not. The final leaf node then provides its assigned label as the return value. The optimization task in this work considers finding trees that optimize a given objective function. In Appendix 9.A.9, we consider arbitrary *separable and totally ordered* objectives (Van der Linden et al., 2023), but in the main text, we limit our discussion for brevity to finding optimal sparse classification trees, for which we need to find the tree that minimizes:

$$C(T, D) = \frac{1}{|D|} \sum_{(x,k)\in D} \mathbb{1}[T(x) \neq k] + \lambda N(T)\,. \tag{9.1}$$

This equation penalizes each misclassification and additionally, the number of leaf nodes $N(T)$ by a complexity cost λ (Hu et al., 2019). Given Eq. (9.1), $T^* = \text{argmin}_{T\in\mathcal{T}(d)} C(T, D)$ finds the optimal tree from the set of all trees $\mathcal{T}(d)$ of maximum depth d.

Rashomon Set Given a Rashomon multiplier ε, the Rashomon set is the set of trees with an objective value within the Rashomon bound $\theta(T^*, D, \varepsilon) = (1 + \varepsilon)C(T^*, D)$. We obtain the Rashomon set:

$$R(T^*, D, \varepsilon) = \{T \in \mathcal{T}(d) \mid C(T, D) \leq \theta(T^*, D, \varepsilon)\}\,. \tag{9.2}$$

Depth-two subroutine A specialized subroutine for finding optimal trees of depth two was proposed by Demirović et al. (2022). It has a significant computation advantage by exploiting precomputed frequency counts instead of repeatedly recursively splitting the dataset. Taking binary classification as example, with D^+ the set of positive samples, then $Q^+(f_i) = |\{(x,k) : (x,k) \in D^+(f_i)\}|$ and $Q^+(f_i, f_j) = |\{(x,k) : (x,k) \in D^+(f_i) \cap D^+(f_j)\}|$ are the number of occurrences of feature f_i, and of f_i and f_j combined respectively in the positive samples, which can be counted efficiently by looping over sparse representations of the feature vectors. The frequency counts for other possible feature inclusions or exclusions of features f_i and f_j in a depth-two tree are calculated implicitly using only these two definitions. E.g., $Q^+(f_i, \bar{f}_j) = Q^+(f_i) - Q^+(f_i, f_j)$ represents the number of instances that satisfy

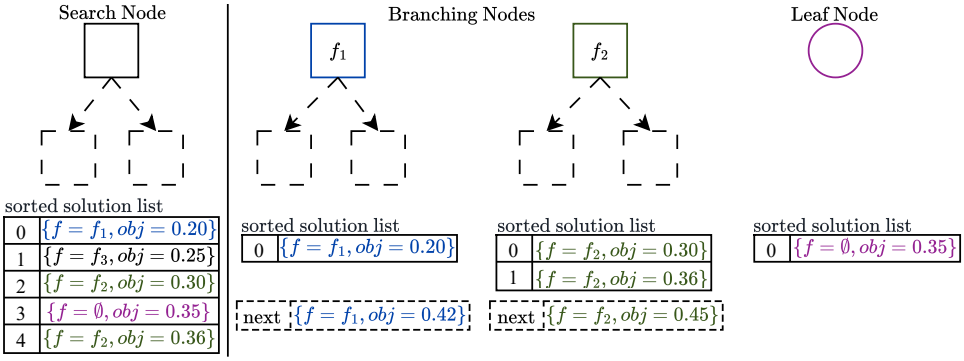


Figure 9.1: Search tree structure. The left-most node is the current search node with its sorted solution list. The middle nodes are branching nodes with features f_1 and f_2. The right-most is a leaf node.

feature f_i, but not f_j. The frequency counts Q^- for negative labels can be calculated analogously. Combining both positive and negative frequency counts allows us to directly compute misclassification scores for all possible depth-two trees.

9.4. In-Order Rashomon Set Calculation

9.4.1. High-level Idea

Given a depth budget, we construct the Rashomon set by exploring decision trees in ascending (i.e., best-first) order of their objective values. We first identify the optimal tree and set the Rashomon bound (tolerance of the best model). We then iteratively build the Rashomon set by maintaining sorted lists of solutions for each node in the search tree. Note that our contribution lies in this second phase: efficiently identifying and ordering these additional solutions.

To compute the Rashomon set, we use a search tree. Each search node maintains a sorted solution list, starting with the optimal one(s), followed by the suboptimal solutions in order of their objective value. A search node contains a helper node for each possible split on all features $f \in F$, and one for creating a leaf node. Each helper node —branching or leaf— of the search node also maintains its own sorted solution list and a pointer to its next unprocessed solution with minimum objective value. Leaf nodes contribute a single solution while branching nodes generate multiple.

In essence, during Rashomon set computation, a search node either repeatedly returns its best next solution from its sorted list or explores new candidates through its helper nodes when no further solution is available. In this exploration phase, the node selects the best next solution among its helpers, and the chosen helper prepares its next candidate if one exists. The search node then returns this selected solution. This process continues until the Rashomon bound or the Rashomon set size is reached. This lazy enumeration strategy computes new solutions only when needed, thereby reducing computation time.

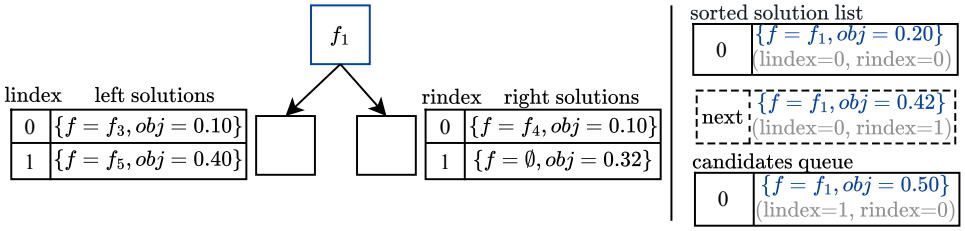


Figure 9.2: Next solution calculation in a branching node.

Example. Fig. 9.1 shows a search node with three helper nodes: two of its branching nodes and one leaf node. Its solution list aggregates the solutions from the helpers. The leaf node's only solution is already included. The branching node with feature f_1 has the next minimum solution value 0.42.

Fig. 9.2 shows how a branching node computes its next solution by combining the best solutions from its left and right child nodes. It computes the Cartesian sum $\{a + b \mid a \in A, b \in B\}$ (see, e.g., (Johnson and Mizoguchi, 1978; Frederickson and Johnson, 1982)), but iteratively and in sorted order, while the sets A and B (the solution lists of the child nodes) at the same time are also iteratively generated (and also in-order). The best solution of this branch node is formed by pairing the child search nodes' best solutions (their solutions with index 0). To find the next-best solution, we increment either the left or right index. The lower-valued combination is selected as the next solution, and the other is stored in the candidates queue. In each step, the next solution is either the minimum valued candidate or a new combination obtained by incrementing one of the indices of the current next solution.

Example. In Fig. 9.2, after adding the best solution of value $0.10 + 0.10 = 0.20$ ($\text{lindex} = 0, \text{rindex} = 0$), two new options with values $0.10 + 0.32 = 0.42$ ($\text{lindex} = 0, \text{rindex} = 1$) and $0.40 + 0.10 = 0.50$ ($\text{lindex} = 1, \text{rindex} = 0$) are considered. The former becomes the next solution, and the latter is stored in the candidates queue.

9.4.2. Algorithm

Phase 1 of our algorithm calculates the optimal tree given a dataset, and a depth budget and meanwhile also caches optimal solutions to subproblems. In Phase 2—the focus of our contribution—we create a search tree over subproblems, with each search node holding a sorted list of computed solutions (so far), and the value of the next best solution (initially set to the cached optimal partial solutions). Phase 2 then iteratively requests the next best solution from the root search node, which recursively constructs it by requesting the next best solutions from its child search nodes (see Appendix 9.A.1).

Alg. 13 shows how a search node computes its next best solution. First, the helper node with the smallest next solution value *node.next* is found (Line 1). A solution group *sol* is then created to collect all solutions with the same objective value (Line 2). If the helper node is a branching node, additional solutions with

the same value are retrieved from its candidates queue CQ (Lines 4-5). This queue is a heap of tuples $(sol.value, lindex, rindex)$, sorted by $sol.value$, while $lindex$ and $rindex$ are solution indices in the left and right child search nodes' solution lists. Using these indices, ExploreCandidates recursively explores each retrieved solution to determine whether incrementing the left or right index produces new solutions with the same objective value (Line 6). Solutions matching the solution group's value are added to the solution group (see Appendix 9.A.2), while the others are added to the candidates queue (Lines 7–8). Then, the next solution value of the branching node becomes the minimum solution value in the candidates queue (Line 9).

Algorithm 13: GetNextSolution()

```
1  node ← arg min_{node∈helper_nodes} node.next
2  sol ← CreateSolutionGroup(node.feature, node.next)
3  if node.IsBranchingNode() then
4  |   same_valued_sols ← node.GetSolutionsWithValue(node.next)
5  |   node.CQ ← node.CQ \ same_valued_sols
6  |   same_valued_sols, others ←
   |     node.ExploreCandidates(same_valued_sols, node.next)
7  |   sol.Add(same_valued_sols)
8  |   node.CQ ← node.CQ ∪ others
9  |   node.next ← node.CQ.Top().value
10 else
11 |   node.CQ ← ∅
12 |   node.next ← ∞
13 if node.CQ = ∅ or node.next > UB then
14 |   helper_nodes ← helper_nodes \ {node}
15 node.SSL ← node.SSL ∪ sol
16 return sol
```

If the helper node is a leaf node, its next solution and candidates queue become empty (Line 11). The helper node is removed from the list of helper nodes if its candidates queue is empty or if the value of its next solution is higher than the search node's upper bound (Lines 12-13). This is because it can no longer contribute new valid solutions in future iterations. Finally, the solution group is added to the helper node's sorted solution list SSL (Line 14), and it is returned as the next solution (Line 15).

Alg. 14 details the ExploreCandidates procedure. The list of solutions to explore is initialized with the same-valued solutions list (Line 2). For each solution in this list, new index pairs are generated by incrementing either the left or right index by one (Lines 6-7). If the resulting index pair has not been evaluated previously (Line 8), the left (right) child search node is queried to return its solution with the left (right) index (Lines 9-10). If the child search node does not have that solution yet, it is obtained by calling **GetNextSolution** (Alg. 13). These left and right solutions are combined into a new solution (Line 11, see Appendix 9.A.3 for the details of the solution combination). If the combined solution has the same value as

the branching node's current next solution, it is added to the same-valued solution list and is explored recursively in later iterations (Lines 13-15). Otherwise, it is added to the distinct-valued solutions list (Lines 16-18).

Algorithm 14: ExploreCandidates($same\_valued\_sols$, $next$)

```
1  distinct_valued_sols ← ∅
2  sols_to_explore ← same_valued_sols
3  while sols_to_explore ≠ ∅ do
4      sol ← sols_to_explore.Pop()
5      for (left_increment, right_increment) ∈ {(1,0),(0,1)} do
6          left_index ← sol.lindex + left_increment
7          right_index ← sol.rindex + right_increment
8          if not Visited(left_index, right_index) then
9              sol_L ← node_L.GetNthSolution(left_index)
10             sol_R ← node_R.GetNthSolution(right_index)
11             sol' ← { node.feature, Combine(sol_L, sol_R) }
12             Visited(left_index, right_index) ← true
13             if sol'.value = next then
14                 same_valued_sols ← same_valued_sols ∪ {sol'}
15                 sols_to_explore ← sols_to_explore ∪ {sol'}
16             else if sol'.value ≤ UB then
17                 sol'.lindex ← left_index
18                 sol'.rindex ← right_index
19                 distinct_valued_sols ← distinct_valued_sols ∪ {sol'}
20 return same_valued_sols, distinct_valued_sols
```

Depth-two subroutine To improve scalability, Alg. 15 adapts the depth-two subroutine proposed by Demirović et al. (2022) (see Section 9.3) to efficiently calculate all depth-two solutions of three branching nodes. It iterates over all pairs of features f_i, f_j (Lines 1-2), with f_i the branching feature in the root, and f_j in either the left or right branching nodes. It then computes the optimal solutions sol_L and sol_R for the left and right subtree (Line 4-5). While doing so, only the solutions with values within the upper bound are kept (Lines 6-9). For each left solution, the combinations obtained with the right solutions are iteratively evaluated and inserted into the node's sorted solution list until the combination exceeds the upper bound (Lines 11-14). See Appendix 9.A.6 for computing trees with one or two branching nodes.

If a limit is set on the Rashomon set size, computing all depth-two trees may be unnecessary. We address this by rerunning the depth-two subroutine with an incrementally increased upper bound until the actual upper bound is reached. See Appendix 9.A.7 for details.

9.4.3. Comparison to the State of the Art

TreeFARMS (Xin et al., 2022) calculates the Rashomon set of trees using a depth-first search and supports only classification tasks. It requires a predefined Rashomon

Algorithm 15: CalculateThreeNodeSols($node, F, Q^+, Q^-$)

```
for f_i ∈ F do
    left_sols ← ∅,   right_sols ← ∅
    for f_j ∈ F, i ≠ j do
        sol_L ←
          Sol(f_j, min{Q+(f̄_i, f_j), Q−(f̄_i, f_j)} + min{Q+(f̄_i, f̄_j), Q−(f̄_i, f̄_j)} + λ)
        sol_R ←
          Sol(f_j, min{Q+(f_i, f_j), Q−(f_i, f_j)} + min{Q+(f_i, f̄_j), Q−(f_i, f̄_j)} + λ)
        if sol_L.value ≤ node.UB then
            left_sols ← left_sols ∪ {sol_L}
        if sol_R.value ≤ node.UB then
            right_sols ← right_sols ∪ {sol_R}
    for left ∈ left_sols  in ascending order do
        for right ∈ right_sols  in ascending order do
            val ← Combine(left, right)
            if val > node.UB then
                break
            node.SSL.Add(Sol(f_i, val))
```

multiplier. Its solutions can be sorted efficiently but only after full enumeration, making correct estimation of the multiplier necessary.

In contrast, SORTD computes the Rashomon set in order of objective value using best-first search, enabling anytime behavior: it produces a valid Rashomon set at any point. This is particularly beneficial for high-dimensional datasets, ensuring search termination and stable performance. It additionally eliminates the need to guess the Rashomon multiplier, as the search can also stop based on a specified set size. When SORTD is run in the same way as TreeFARMS (i.e., with a fixed Rashomon bound), SORTD returns the whole Rashomon set up to two orders of magnitude faster than TreeFARMS, while using up to one order of magnitude less memory, as we will show in the next section. Furthermore, SORTD is not limited to classification tasks, supporting any separable and totally ordered objective for computation and any separable and partially ordered objective for post-evaluation.

Limitations Our approach is limited to binary features, which is a common limitation (e.g., Xin et al., 2022). And since finding even a single optimal tree is NP-hard, enumerating the Rashomon set for larger depth budgets, dataset sizes, or target set sizes becomes intractable. However, our scalability improvements extend the range of problem instances that can practically be solved.

9.5. Experimental Evaluation

We conduct a series of experiments with the following aims: (1) to assess SORTD's runtime efficiency in computing Rashomon sets; (2) to showcase that a small number of high-quality trees—easily found by SORTD—may be informative for model

evaluation via variable importance analysis; and (3) to demonstrate SORTD's flexibility in enumerating and analysing Rashomon sets under varying objective functions.

Experiment set-up For aims (1) and (2), we use the 30 benchmark binary classification datasets previously used to assess state-of-the-art methods (Narodytska et al., 2018; Verwer and Zhang, 2019; Aglin et al., 2020a; Demirović et al., 2022; Xin et al., 2022). For aim (3) we adopt common regression (Zhang et al., 2023) and fairness benchmark datasets (Le Quy et al., 2022). We implemented SORTD in C++ and provide it as a python package.[1] We use STreeD (Van der Linden et al., 2023) to compute optimal trees in SORTD's first phase. All experiments are run single-threaded on an Intel Xeon E5-6448Y @ 2.1 GHz with 100 GB RAM, with a 300 seconds time limit. Further details are provided in Appendix 9.B.

9.5.1. Runtime Performance

We evaluated SORTD's runtime performance against the state-of-the-art method TreeFARMS (Xin et al., 2022) across varying Rashomon set sizes n^T. Since TreeFARMS requires the Rashomon multiplier (or bound) to be specified in advance, we ensured a fair comparison by precomputing the Rashomon multipliers (see Appendix 9.B.2) using the following procedure. We varied the depth budget $d \in \{3, 4, 5\}$ and the complexity cost $\lambda \in \{0.001, 0.01, 0.1\}$. Using each (dataset, d, λ) combination, we ran SORTD to find the smallest multiplier ε that yields at least $n^T = 10^n$ trees for $n \in \{1, \ldots, 6\}$. We limited n to six, as we consider sets larger than 10^6 trees to be impractical for real-world analysis. Both methods then used these multipliers for Rashomon set enumeration, and runtime was measured to obtain the *whole* Rashomon set up to the specified bound.

Fig. 9.3 plots the empirical cumulative distribution of runtimes for finding the whole Rashomon set of the specified size, aggregated over all datasets and λ values (see Appendix 9.B.2 for detailed results). Across every depth budget and Rashomon set size, SORTD consistently outperforms TreeFARMS, reaching speed-ups of up to two orders of magnitude. As the depth budget grows, SORTD's runtime increases only slightly—most sets are produced under 10 seconds even at the largest depth—whereas TreeFARMS slows by more than an order of magnitude and hits the time limit after depth budget three. Moreover, at higher depth limits, SORTD scales better for increasingly large Rashomon sets than TreeFARMS.

We additionally investigated how feature dimensionality affects runtime. Fig. 9.4 shows the empirical cumulative distribution of runtimes at depth budget four and $n^T = 10^6$ while varying the feature dimension of the input dataset. Runtime increases for both methods as dimensionality grows, but SORTD is impacted less. Once the feature count exceeds 30, TreeFARMS fails to finish computing some Rashomon sets within the time limit. In contrast, SORTD remains fast: even with 30 features, it completes all runs well below the time limit.

Furthermore, we evaluated the memory usage of SORTD. Fig. 9.5 shows the empirical cumulative distribution of memory usage in gigabytes. For most of the

[1] `https://github.com/ConSol-Lab/pysortd`

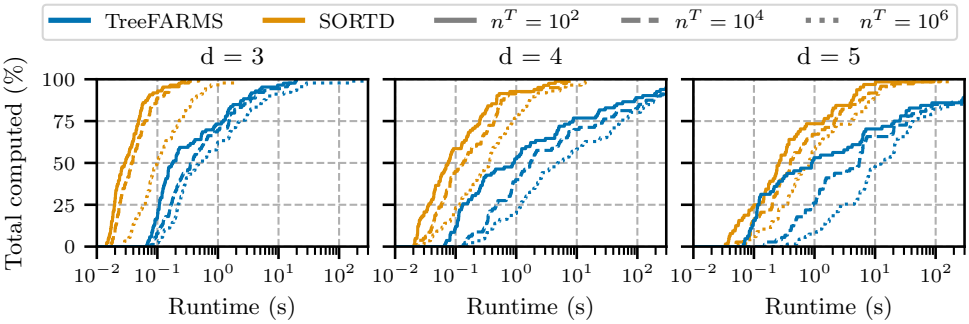


Figure 9.3: Cumulative runtime (s) distribution across tree depths d and Rashomon set sizes n^T. The x-axis is logarithmic and shows the runtime for enumerating the full Rashomon set. SORTD is up to two orders of magnitude faster than TreeFARMS.

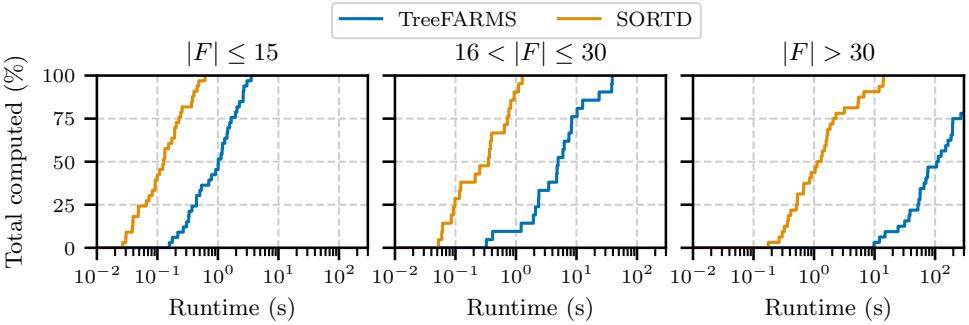


Figure 9.4: Cumulative runtime (s) distribution across varying feature dimensionality, with depth budget four and $n^T = 10^6$. The x-axis is logarithmic and shows the runtime for enumerating the full Rashomon set. SORTD scales better with more features than TreeFARMS.

instances, SORTD's memory usage remains below 1 GB, leading to on order of magnitude less memory usage. In contrast, TreeFARMS shows substantially higher memory requirements, particularly at greater depths.

9.5.2. Variable Importance Analysis

To test whether SORTD's in-order solution generation facilitates downstream evaluation, we measured variable importance with the *Leave-One-Feature-Out* (LOFO) score (Lei et al., 2018) by computing the increase in the area under the Rashomon objective curve when that feature is omitted. Fig. 9.6 illustrates LOFO on the *compas* (Angwin et al., 2016a) and *fico* (FICO et al., 2018) datasets using the top-100 trees. As reported by Fisher et al. (2019), "priors > 3" is an important variable in *compas*. In our analysis, omitting this feature noticeably shifts the loss curve, and a similar effect is observed when excluding external risk estimate features in *fico*, highlighting their strong predictive influence.

We now start our test whether variable importance obtained through different

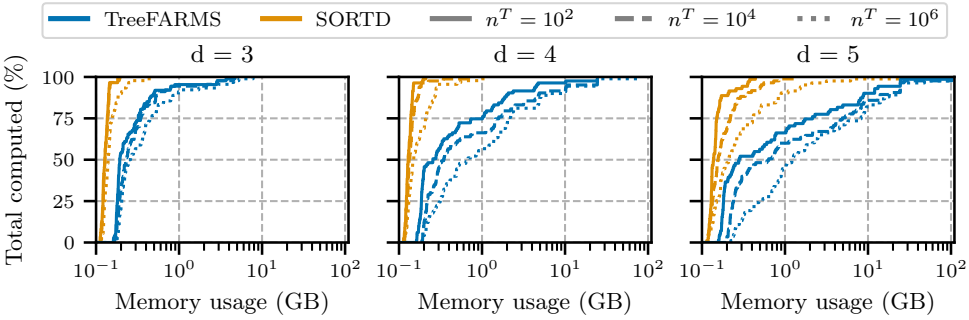


Figure 9.5: Cumulative memory usage (GB) distribution across tree depths d and Rashomon set sizes n^T. Note the logarithmic x-axis. SORTD uses one order of magnitude less memory than TreeFARMS.

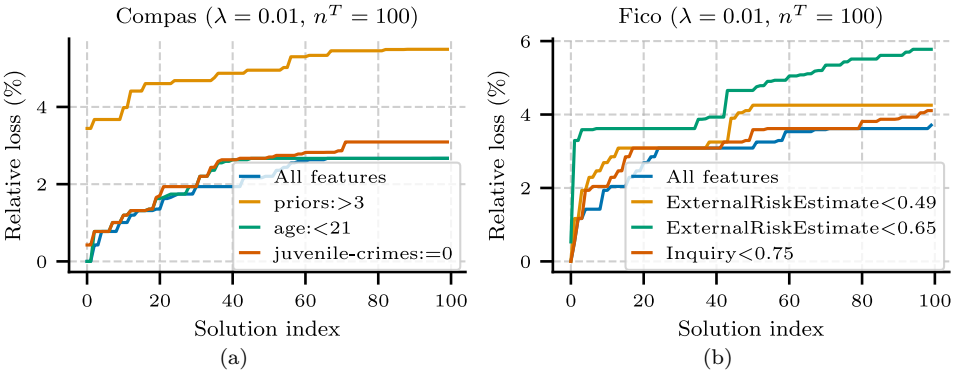


Figure 9.6: Top-3 influential features identified via LOFO. Shifts in the relative loss curves after removing each feature indicate their influence on the Rashomon set for the *compas* and *fico* datasets.

Rashomon set sizes provides similar insights. We treated variable importance values derived from the top-10,000 trees as a reference, and compared them with the ones obtained from the top-1 and top-100 trees. Since importance estimates within Rashomon sets can vary under resampling (Donnelly et al., 2023), each dataset was bootstrapped 20 times. For each resample, we computed the Rashomon multiplier required to obtain at least 10,000 trees and constructed the corresponding set using the multipliers. We then evaluated the increase in area under the Rashomon objective curve using $\lambda = 0.01$ and a depth budget of four. Due to its high runtime requirement, the *biodeg* dataset was not used in this experiment.

We evaluated top-5 feature stability using the *Jaccard index* ($|A \cap B|/|A \cup B|$) and full ranking stability using *Kendall's* τ (Kendall, 1938). Comparing top-1 to top-10,000 trees, 19 of 29 datasets had a Jaccard index ≥ 0.8; this increased to 26 datasets for top-100. Similarly, $\tau \geq 0.7$ held for 5 datasets using top-1 trees, and for 17 using top-100 trees (see Appendix 9.B.7). These results suggest that variable importance is relatively stable when using 100 trees, highlighting the potential of estimating

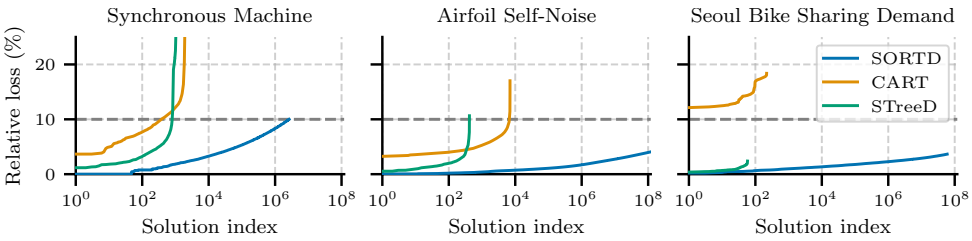


Figure 9.7: Relative loss compared to the optimal solution of all regression trees found within the one minute time-out for max-depth four, $\lambda = 0.001$, and $\varepsilon = 0.1$. Dashed lines indicate the Rashomon bound. SORTD finds orders of magnitude more trees in the Rashomon set than the other methods.

the importance values from smaller Rashomon sets. This is particularly useful for large datasets, where computing large Rashomon sets is impractical. Developing theoretical guidelines for the minimum number of trees required to obtain stable importance estimates could further facilitate this analysis.

9.5.3. Evaluating Other Objectives

SORTD also supports objectives beyond accuracy (see Appendix 9.A.9), optimizing separable totally ordered objectives directly and evaluating separable and partially ordered ones indirectly.

Regression For example, SORTD can directly find the Rashomon set of sparse regression trees with the totally ordered objective of minimal mean-squared error. We demonstrate this capability by comparing SORTD with the heuristic CART (Breiman et al., 1984) and the optimal method STreeD (Van der Linden et al., 2023) since no other method is known for generating this set. Both methods repeatedly compute trees for random samples of the data until a given time limit. Fig. 9.7 shows that SORTD finds trees in order until the 10% relative bound is reached, whereas, as observed by Xin et al. (2022), for classification tasks, CART and STreeD find orders of magnitude fewer trees in the Rashomon set. See Appendix 9.B.8 for further details.

Equality of opportunity SORTD can also evaluate partially ordered objectives such as multi-objective criteria indirectly. E.g., to find fair accurate trees, SORTD can first obtain the Rashomon set based on accuracy, and then evaluate it using another objective such as *equality-of-opportunity*, i.e., the difference in the true positive rate of two discrimination-sensitive groups. For example, Fig. 9.8 shows the top 10^7 trees in the Rashomon set for accuracy evaluated with the equality-of-opportunity metric. In Appendix 9.B.9 we provide further details and a runtime comparison with STreeD.

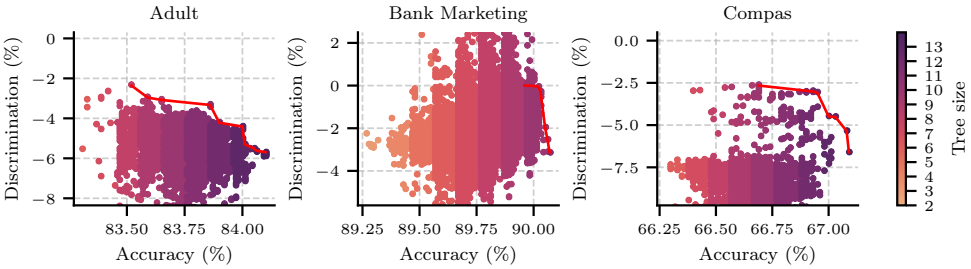


Figure 9.8: The top 10^7 trees in the accuracy Rashomon set evaluated using *equality-of-opportunity* with max-depth four and $\lambda = 0.001$. The red lines show the Pareto-front of accuracy and fairness. The colours indicate tree size. The sign of the discrimination shows which group is disadvantaged.

9.6. Conclusion

The Rashomon set of sparse decision trees offers many advantages over relying on a single model, provided it can be explored efficiently. SORTD makes such exploration practical: it enumerates trees in ascending objective order, allowing the highest-quality candidates to be retrieved and evaluated quickly. At the same time, its algorithmic design scales significantly better than the state of the art and uses less memory as either the feature dimensionality or the depth budget grows. Finally, it supports separable totally ordered objectives for Rashomon set computation, as well as separable and partially ordered objectives for fast post-hoc evaluation. Together, these advances turn large-scale Rashomon-set analysis and model selection into a viable option for real-world applications.

Appendices for Chapter 9

9.A. Detailed Method Description

9.A.1. Main Algorithm of Rashomon Set Calculation

We generate the Rashomon set by iteratively computing trees in ascending order of their objective values. Alg. 16 summarizes how the Rashomon set is computed once the optimal solution has been found. The algorithm takes as input: the maximum depth d, the training dataset D, the optimal solution value sol^*, the Rashomon multiplier ε, the maximum number of trees n^T, and a cache of optimal solutions of the subtrees. The Rashomon multiplier and the maximum number of trees are complementary: specifying either one is sufficient. If the Rashomon multiplier is omitted, a large default value is used to derive the bound.

Next, the root search node is initialized by setting its upper bound to the Rashomon bound and retrieving its optimal solution from the cache (Line 2). This optimal value is then compared with the bound to determine whether any solutions lie within the Rashomon set. If the condition is satisfied, additional solutions are iteratively added to the node's sorted solution list until a stopping criterion is met (Lines 4–6).

Algorithm 16: Main($d, D, sol^*, \varepsilon, n^T, cache$)

```
1 θ ← sol*.value × (1 + ε)
2 root_search_node ← InitializeSearchNode(d, D, θ, cache)
3 sol ← root_search_node.sol*
4 while sol.value ≤ θ and |root_search_node.SSL| < n^T do
5 |   sol ← root_search_node.GetNextSolution()
6 |   root_search_node.SSL ← root_search_node.SSL ∪ {sol}
7 return root_search_node.SSL
```

9.A.2. Solutions Structure

A Rashomon set may contain well over a billion trees, and enumerating such a large set has a high computational and memory load. Therefore, we adopt the grouped solution structure used in (Xin et al., 2022): Fig. 9.9 shows how (partial) solutions (i.e., subtrees) are stored in memory. Solutions are recursively grouped by their objective value and the splitting feature of the (subtree) root node, followed by a list of pairs of solutions for the left and right subtrees. The details of how same-valued solutions are grouped are given in Section 9.4.2.

Example. In Fig. 9.9, the left of the figure represents different solution values of a branching node. Because the first two solution values are the same, a solution group (right upper side of the image) with two solution pairs is created. As for the remaining solution value, a solution group (right lower side of the image) with one solution pair is created.

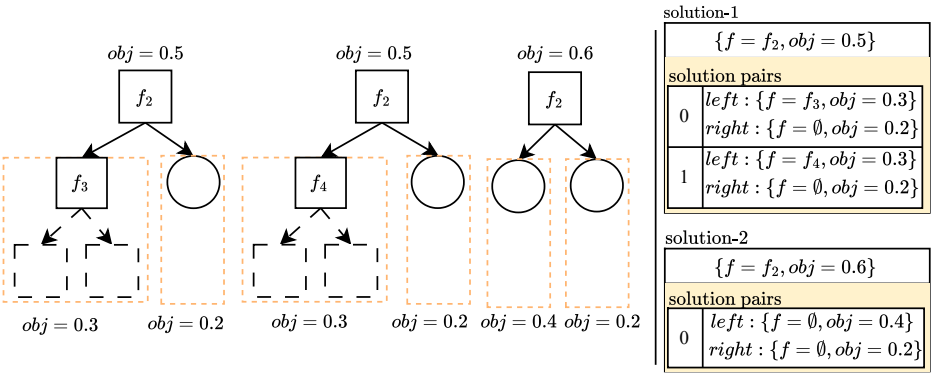


Figure 9.9: SORTD grouped solution structure.

9.A.3. Incorporating the Regularization Cost

The regularization term in the objective function penalizes model complexity by adding λ for every leaf node (Eq. (9.1)). Consequently, creating a new split that raises the leaf node count by one increases the objective value even if predictive accuracy is unchanged. To capture this effect, we assign a fixed cost of λ to each branching decision. If the tree consists of a single root leaf, the same cost λ is added once for that leaf node.

We use the branching cost while combining left and right solutions and while determining the upper bounds of the child search nodes (see Appendix 9.A.4).

To construct a solution for a branching (search) node, the algorithm first calculates solutions for its left and right child nodes. It then combines these partial solutions to obtain the parent solution (Alg. 14, Line 11). Since the learning objective imposes a regularization penalty of λ per additional leaf, we add the same branching cost of λ when combining the left and right subtree solutions as follows:

$$\text{Combine}(sol_L, sol_R) = sol_L.value + sol_R.value + \lambda\,. \tag{9.3}$$

9.A.4. Upper Bounding

To avoid exploring low–quality regions that cannot contribute to the Rashomon set, every search node is equipped with an upper bound on the solution value of any tree it may still yield. The root node's upper bound is initialized with the Rashomon bound, and each helper node simply inherits its parent's upper bound. Child search nodes, however, receive tighter bounds that account for the best solution already attainable in the complementary subtree. Given a branching node and its upper bound UB, the upper bounds of the child search nodes are calculated as follows:

$$\text{UB}_\text{L}(UB, sol_R^*) = UB - sol_R^* - \lambda\,, \tag{9.4}$$

$$\text{UB}_\text{R}(UB, sol_L^*) = UB - sol_L^* - \lambda\,. \tag{9.5}$$

Here, λ is to account for the branching cost (see Appendix 9.A.3) while sol_L^* and sol_R^* are the optimal solution values of the left and right child search nodes, respectively.

These optimal solutions are either retrieved from a cache of optimal solutions or computed using STreeD (Van der Linden et al., 2023) if absent.

We also use the upper bounds of branching and search nodes for two additional purposes: (1) to filter out helper nodes whose next solution value exceeds its search node's upper bound (see Alg. 13); and (2) to determine whether a newly generated solution obtained by combining a left and a right solution should be accepted, by comparing it against the branching node's upper bound (see Alg. 14).

9.A.5. Caching

A subtree can be encountered multiple times during the search. To avoid recomputing its solutions in different parts of the search space, we cache both solutions themselves—stored in the search node's sorted solution list—and the objects required to compute them, namely the branching search nodes.

We adopt the caching mechanism of Demirović et al. (2022), which supports two strategies: *dataset caching*, in which a subtree is identified by the set of samples it contains, and *branch caching*, in which it is identified by the features used in every branching decision from the root to that subtree. In our experiments, we used dataset caching.

When a new search node is created, the algorithm first queries the cache. If the subtree is present, the node reuses the cached solution list and branching nodes. The solution list is then extended—if not already exhausted—by all search nodes that share it.

9.A.6. Depth-Two Subroutine

We use a specialized algorithm to calculate the Rashomon set when the remaining depth budget is at most two. Section 9.4.2 details how, for a given search node, we enumerate all trees that contain three branching nodes. Below, we describe the cases with one and two branching nodes.

Alg. 17 outlines the procedure for generating all trees with a single branching node. The algorithm takes as input a search node *node*, a feature set F, class frequency counts Q^+ and Q^-, and an upper bound UB. For each feature in F, the algorithm computes the label assignments for the left and right children that minimize their respective subtree costs. These two partial solutions are then combined (Line 2; see Appendix 9.A.3 for details). If the resulting solution value is within the upper bound, the solution is added to the search node's sorted solution list (Lines 3-4).

Algorithm 17: CalculateOneNodeSol(*node*, F, Q^+, Q^-, UB)

```
1 for f_i ∈ F do
2 |   val ← Combine(min{Q⁺(f̄_i), Q⁻(f̄_i)}, min{Q⁺(f_i), Q⁻(f_i)})
3 |   if val ≤ UB then
4 |   |   node.SSL.SortedInsert(Sol(f_i, val))
```

For trees with exactly two branching nodes, the feasible topology is restricted to one of two mirror images: either the left subtree is a leaf and the right subtree

contains a single branching node, or vice versa. Alg. 18 handles the case in which the right child is a leaf; the symmetric case is treated analogously.

The procedure begins by selecting a feature for the root split and computing the solution of the right subtree being a leaf (Line 3). Next, it creates a second split in the left subtree using any remaining feature from F and evaluates every possible label assignment, selecting the minimal left-subtree solution value (Line 5). If this value does not exceed the upper bound, the left-subtree solution is stored (Lines 6-7). Each stored left solution is then combined with the right-leaf solution. The resulting trees are inserted into the search node's sorted solution list whenever its overall solution value remains within the upper bound (Lines 8-12).

Algorithm 18: CalculateTwoNodeSols($node$, F, Q^+, Q^-, UB)

```
1  for f_i ∈ F do
2  |  left_sols ← ∅
3  |  val_R ← min{Q⁺(f_i), Q⁻(f_i)}
4  |  for f_j ∈ F, i ≠ j do
5  |  |  val_L ←
   |  |    Combine(min{Q⁺(f̄_i, f_j), Q⁻(f̄_i, f_j)}, min{Q⁺(f̄_i, f̄_j), Q⁻(f̄_i, f̄_j)})
6  |  |  if val_L ≤ UB then
7  |  |  |  left_sols ← left_sols ∪ {Sol(f_j, val_L)}
8  |  for left ∈ left_sols do
9  |  |  val ← Combine(left.value, val_R)
10 |  |  if val ≤ UB then
11 |  |  |  sol ← UpdateSolution(f_i, val)
12 |  |  |  node.SSL.SortedInsert(sol)
```

9.A.7. Gradual Solution Creation

When the Rashomon set is subject to a size limit (see Alg. 16), enumerating only a subset of the depth-two trees can suffice. We achieve this by tightening the upper bound, hence pruning more candidates, reducing memory usage and runtime. If the bound is too low, however, the algorithm may spend significant effort (e.g., computing frequency counts) only to yield a single solution; if it is too high, many unused trees are still generated. Empirically, a bound that guarantees the inclusion of the leaf solution and all trees with at most one branching node strikes a good balance. Therefore, we initialize the depth-two upper bound as

$$\hat{UB} \;=\; \min\big\{\, \mathrm{C}(T, D) + \lambda,\; UB \big\}\,, \tag{9.6}$$

where D and UB denote the dataset and the search node's bound, T is the single-leaf tree at that node, and λ is the branching cost (Appendix 9.A.3).

With $\hat{UB}$ set, Alg. 15, 17, and 18 generate the depth-two solutions. If the calculated solutions are exhausted before the Rashomon-set size limit is reached, we

relax the bound as follows:

$$\hat{UB} \leftarrow \begin{cases} UB & (UB - \hat{UB}) < 0.2 \cdot UB \\ \hat{UB} + 0.5\,(UB - \hat{UB}) & \text{otherwise} \end{cases} \tag{9.7}$$

This increases the upper bound at a diminishing rate, as the number of solutions grows exponentially with respect to the changes in the upper bound.

9.A.8. Ignoring Trivial Extensions

By default, SORTD allows extending a tree node with two child nodes that share the same label. Following Xin et al. (2022), we refer to these as *trivial extensions*. SORTD can optionally disregard such trees when calculating the Rashomon set. A trivial extension occurs when a branching node produces two leaf nodes whose labels are identical, so the branching does not change the objective value except for the added complexity cost. A trivial extension is detected during the depth-two subroutine (e.g., Alg. 15 Line 12) or branch-tracker iterations (Alg. 14 Line 11) if the following condition holds:

$$\text{IsLeaf}(\textit{left}) \textbf{ and } \text{IsLeaf}(\textit{right}) \textbf{ and } \textit{left.label} = \textit{right.label} \tag{9.8}$$

In some cases, at least one leaf label may be arbitrary: assigning one label or another does not affect the objective value. In this case, SORTD assigns one label arbitrarily, stores one of the alternatives, and when combining with another leaf node, selects the label that avoids creating a trivial extension.

Removing trivial extensions reduces the number of trees in a Rashomon set for a fixed Rashomon multiplier, or equivalently, increases the Rashomon multiplier attainable for a fixed set size. As shown in Appendix 9.B.6, SORTD maintains performance improvements of up to two orders of magnitude also when trivial extensions are excluded.

9.A.9. Optimizing Other Objectives

SORTD supports computing Rashomon sets for separable and totally ordered objectives *directly* and post-evaluating other separable and partially-ordered objectives over an already computed Rashomon set *indirectly*.

Optimizing totally ordered objectives SORTD's Rashomon construction is based on the STreeD framework (Van der Linden et al., 2023), and therefore easily generalizes to other objectives than the regularized misclassification score. However, since SORTD requires sorting partial solutions in its search nodes, it cannot support all optimization tasks that STreeD supports i.e., all separable and partially ordered tasks. Specifically, SORTD requires *separable and totally ordered* solutions. Additionally, in our implementation, we assume all objectives are *additive*, i.e., solutions from left and right subtrees can be combined using addition. SORTD therefore, has the same generalizability as DL8 (Nijssen and Fromont, 2010).

To support other objectives, the only major change is to redefine the objective function $C(T, D)$. For example, to optimize regression trees by minimizing the *sum*

of squared errors, we define:

$$C(T, D) = \sum_{(x,y)\in D} (T(x) - y)^2 + \lambda N(T)\,. \tag{9.9}$$

To illustrate this, we show in Appendix 9.B.8 SORTD's performance in finding the Rashomon set for regression trees.

Post-evaluating other objectives After generating a Rashomon set, the trees in this set can be evaluated using any objective by going over each tree one at a time. However, computing an objective for each tree individually can be expensive. Therefore, we provide a special procedure to post-evaluate objectives that are *separable* (as defined by Van der Linden et al. (2023)), i.e., we can compute the objective values separately for the left and right subtrees and combine them afterwards. This requires defining some cost function $g(D, k)$ for leaf nodes with D the input dataset and k the assigned label. This cost function does not necessarily need to return a real-valued number, but can also, for example, return a tuple of values. Similarly, we need to define a combining operator $\oplus$ that takes a left and right solution and returns the cost of combining both (again, not necessarily a real-valued number).

SORTD uses these functions to construct a new Rashomon set, with equal-valued solutions grouped together (see Appendix 9.A.2). However, in this post-evaluation, we no longer require the objective to be totally ordered, so we can no longer acquire the solutions in order of the second objective. In practice, we obtain the Rashomon set in order according to the optimization objective (e.g., regularized accuracy), by retrieving them in batches. We run Alg. 16 for a certain Rashomon set size; evaluate the obtained solutions using the second objective; and possibly rerun Alg. 16 with a larger Rashomon set size, continuing where we left. This can be repeated until a certain time-out or some other user-defined stopping criterion.

To facilitate ease of evaluation, SORTD allows users to define the leaf cost function g and the combining operator $\oplus$ in Python and pass them to SORTD. Consider, for example, optimizing trees with a multi-objective criterion of regularized accuracy and the fairness metric *equality-of-opportunity*. The equality of opportunity loss is defined as the difference in the true positive rate between two discrimination sensitive groups. More formally, let y be the true label, $\hat{y}$ the predicted label, then equality of opportunity requires:

$$P(\hat{y} = 1 \mid y = 1, \text{group one}) = P(\hat{y} = 1 \mid y = 1, \text{group two})\,. \tag{9.10}$$

We can compute this by defining a leaf cost function that counts for both groups how many samples with the true positive label are also predicted with a positive label. Additionally, we count the total number of misclassifications, so that we can also compute the accuracy. Hence, the solution tuple returned will be $(\mathit{misclassifications}, \mathit{positive\_count\_group\_0}, \mathit{positive\_count\_group\_1})$. The combining operator is then simply element-wise addition of two tuples. Finally, we need to provide a function that computes from these tuples in the root node the actual objective value. In this example, the final objective value is a tuple of the accuracy

and the difference in the true positive rate for the two groups. These tuples can then be filtered to only keep the Pareto front. In Python, we can write:

```
def leaf(dataview, label):
  mis = dataview.num_instances_for_label(1 - label)
  if label == 1:
    # Assume feature 0 is the discrimination sensitive feature
    group1_size = dataview.num_instances_for_label_and_feature(1, 0)
    group0_size = dataview.num_instances_for_label(1) - group1_size
    return (mis, group0_size, group1_size)
  return (mis, 0, 0)

def add(sol1: tuple, sol2: tuple):
  return (sol1[0] + sol2[0], sol1[1] + sol2[1], sol1[2] + sol2[2])

def obj(sol: tuple):
  acc  = 1.0 - sol[0] / N
  disc = (sol[1] / N_pos_group0) - (sol[2] / N_pos_group1)
  return (acc, disc)

model = SORTDClassifier("cost-complex-accuracy", max_depth=3, cost_complexity
    =0.01, max_num_trees = 10000)
model.fit(X, y)
results = model.evaluate_other_objective(X, y, leaf, add)
solution_values = [obj(s.objective) for s in results]
```

Listing 9.1: Example Python Code for optimizing accuracy and equality of opportunity.

9.B. Experiment Details

9.B.1. Classification Datasets

We selected all 46 binary classification datasets with less than 100 binary features from Demirović et al. (2022) and Xin et al. (2022).[2] We excluded nine duplicate datasets and three trivially solvable ones (solved in under one second). We also removed four datasets that exceeded memory limits when calculating the Rashomon multipliers for specific Rashomon set sizes. This computation was memory intensive because the Rashomon multipliers were unknown, which required setting the upper bound to a high value.

After these exclusions, 30 datasets remained. From these, we eliminated duplicate and complementary features. Table 9.1 lists the datasets together with their sample size and resulting feature size, $|D|$ and $|F|$, respectively. These datasets were used in runtime (Section 9.5.1) and variable importance analysis (Section 9.5.2). The original datasets can be obtained from the UCI Machine Learning repository (Dua and Graff, 2017) and from Bessiere et al. (2009), Angwin et al. (2016a), Wang et al. (2017), and FICO et al. (2018).

9.B.2. Runtime Performance

We generated benchmark instances by varying the complexity penalty $\lambda \in \{0.001, 0.01, 0.1\}$ and the depth budget $d \in \{3, 4, 5\}$. For each (dataset, λ, d) combination, we use SORTD to compute the minimum Rashomon multiplier (and corresponding Rashomon bound) that yields at least 10^n trees for every $n \in \{1, \ldots, 6\}$. Because

[2] `https://bitbucket.org/EmirD/murtree`, `https://github.com/ubc-systopia/treeFarms`

Dataset	$\|D\|$	$\|F\|$	Dataset	$\|D\|$	$\|F\|$
anneal	812	44	HTRU2	17898	57
bank	4521	23	hypothyroid	3247	39
banknote	1372	16	kr-vs-kp	3196	38
bar-7	1913	14	lymph	148	47
biodeg	1055	81	messidor	1151	24
breast	699	10	monk1	124	15
car	1728	15	monk2	169	11
cheap	2653	15	monk3	122	15
coffee	3816	15	mouse	70	45
compas	6907	12	soybean	630	42
diabetes	768	11	spect	267	22
expensive	1417	15	tic-tac-toe	958	18
fico	10459	17	tumor	336	17
haberman	306	92	vote	435	48
hepatitis	137	34	yeast	1484	46

Table 9.1: Classification datasets

both methods employ a grouped solution structure, requesting 10^n solutions may sometimes yield more than 10^{n+1} solutions, with the additional solutions sharing the same objective value. In such cases, we did not generate a separate instance for the 10^{n+1} target, as it was already satisfied. Finally, we ran each instance five times.

We compare SORTD with the state-of-the-art algorithm TreeFARMS (Xin et al., 2022), under a 300-second time limit per instance. Each experiment is repeated five times. We set `rashomon_ignore_trivial_extensions = False` for both methods (see Section 9.B.6 for runtime results with trivial extensions ignored). Runtime results are presented in Tables 9.2-9.4. The results of the instances with depth budget three are not presented as both methods solved most of them within a second. Entries marked '–' indicate omitted runs due to excessive memory usage. We denote runtimes below one second with '< 1', and timeouts with '> 300'.

As expected, increasing λ simplifies the search problem for both methods. A larger penalty on leaf usage encourages shallower trees, which effectively prunes deeper parts of the search space. In contrast, small λ values lead to significantly larger search spaces.

Across most instances, SORTD completes enumeration within one second for depth four and within ten seconds for depth five. In contrast, TreeFARMS exhibits high variability—even for depth four, runtimes range from below a second to full timeouts. Its performance degrades further with increasing depth, frequently timing out for $\lambda = 0.001$ and $\lambda = 0.01$ especially when the feature size of the dataset is high.

To evaluate the overall performance of SORTD compared to TreeFARMS, Tables 9.2-9.4 also report the geometric mean of the average runtime ratios $t_{\text{TreeFARMS}}/t_{\text{SORTD}}$, where timeout values are considered as 300 seconds. Results show that SORTD achieves up to two orders of magnitude improvement in runtime over Tree-

			$d = 4$		$d = 5$	
Dataset	$\|D\|$	$\|F\|$	TreeFARMS	SORTD	TreeFARMS	SORTD
breast	699	10	<1	<1	<1	<1
monk2	169	11	<1	<1	2±0.1	<1
diabetes	768	11	<1	<1	<1	<1
compas	6907	12	<1	<1	1±0.0	<1
bar-7	1913	14	<1	<1	1±0.0	<1
car	1728	15	<1	<1	5±0.2	<1
expensive	1417	15	<1	<1	6±0.1	<1
cheap	2653	15	1±0.0	<1	6±0.1	<1
monk1	124	15	<1	<1	4±0.7	<1
monk3	122	15	1±0.0	<1	22±0.7	<1
coffee	3816	15	1±0.0	<1	7±0.2	<1
banknote	1372	16	<1	<1	<1	<1
tumor	336	17	2±0.1	<1	20±0.8	<1
fico	10459	17	5±0.2	<1	32±1.1	2±0.0
tic-tac-toe	958	18	2±0.0	<1	21±0.0	<1
spect	267	22	22±0.7	<1	>300	<1
bank	4521	23	8±0.1	<1	88±0.4	2±0.0
messidor	1151	24	5±0.3	<1	33±0.7	<1
hepatitis	137	34	103±2.6	<1	>300	61±14.8
kr-vs-kp	3196	38	46±0.6	<1	>300	12±0.1
hypothyroid	3247	39	49±0.8	<1	>300	9±0.5
soybean	630	42	57±2.0	<1	>300	7±0.2
anneal	812	44	24±0.3	<1	>300	5±0.1
mouse	70	45	9±0.1	<1	>300	5±0.1
yeast	1484	46	184±11.3	<1	>300	14±0.4
lymph	148	47	139±6.1	<1	>300	9±0.0
vote	435	48	>300	<1	>300	13±0.1
HTRU2	17898	57	>300	5±0.2	>300	140±2.5
biodeg	1055	81	>300	4±0.0	>300	203±5.9
haberman	306	92	>300	2±0.0	>300	155±3.8
Geometric mean improvement				86.95		24.30

Table 9.2: Runtime (s) performance of methods to calculate Rashomon sets across datasets and depth budgets with $\lambda = 0.001$ and $n^T = 10^n$, $n \in \{1, .., 6\}$. Results are reported as mean ± standard error. A '–' indicates results that are omitted due to their high memory requirement.

FARMS for Rashomon set enumeration. These findings further emphasize SORTD's scalability, particularly as problem complexity increases with respect to tree depth, dataset size, and regularization parameter λ.

Longer runtime evaluation for timeout instances Runtime results in Tables 9.2-9.4 show that several TreeFARMS (Xin et al., 2022) instances reached the 300-second time limit. To further examine these cases, we extended the runtime limit to 3600 seconds. Tables 9.5-9.6 present the results for depth budgets four and five, respectively.

			$d = 4$		$d = 5$	
Dataset	$\|D\|$	$\|F\|$	TreeFARMS	SORTD	TreeFARMS	SORTD
breast	699	10	<1	<1	<1	<1
monk2	169	11	<1	<1	2±0.1	<1
diabetes	768	11	<1	<1	<1	<1
compas	6907	12	<1	<1	<1	<1
bar-7	1913	14	<1	<1	1±0.0	<1
car	1728	15	<1	<1	5±0.2	<1
expensive	1417	15	<1	<1	6±0.1	<1
cheap	2653	15	1±0.1	<1	5±0.4	<1
monk1	124	15	<1	<1	5±1.1	<1
monk3	122	15	2±0.1	<1	32±1.5	<1
coffee	3816	15	1±0.1	<1	7±0.3	<1
banknote	1372	16	<1	<1	1±0.0	<1
tumor	336	17	2±0.1	<1	19±0.5	<1
fico	10459	17	5±0.2	<1	37±1.6	3±0.1
tic-tac-toe	958	18	2±0.0	<1	21±0.1	<1
spect	267	22	33±1.2	<1	>300	<1
bank	4521	23	7±0.4	<1	49±5.8	3±0.2
messidor	1151	24	5±0.4	<1	33±2.8	<1
hepatitis	137	34	125±2.4	<1	>300	72±9.2
kr-vs-kp	3196	38	46±0.9	<1	>300	11±0.2
hypothyroid	3247	39	21±3.5	<1	79±20.3	6±0.4
soybean	630	42	102±10.0	<1	>300	5±0.2
anneal	812	44	25±0.7	<1	>300	5±0.2
mouse	70	45	11±0.3	<1	>300	5±0.1
yeast	1484	46	189±10.7	<1	>300	13±0.7
lymph	148	47	241±13.9	<1	>300	6±0.1
vote	435	48	>300	<1	>300	8±0.5
HTRU2	17898	57	198±24.7	8±0.4	224±24.9	117±6.0
biodeg	1055	81	>300	4±0.1	-	-
haberman	306	92	>300	3±0.3	>300	133±2.7
Geometric mean improvement				68.57		24.60

Table 9.3: Runtime (s) performance of methods to calculate Rashomon sets across datasets with $\lambda = 0.01$ and $n^T = 10^n$, $n \in \{1, .., 6\}$. Results are reported as mean ± standard error. A '–' indicates results that are omitted due to their high memory requirement.

			$d = 4$		$d = 5$	
Dataset	$\|D\|$	$\|F\|$	TreeFARMS	SORTD	TreeFARMS	SORTD
breast	699	10	<1	<1	<1	<1
monk2	169	11	<1	<1	1±0.3	<1
diabetes	768	11	<1	<1	<1	<1
compas	6907	12	<1	<1	<1	<1
bar-7	1913	14	<1	<1	<1	<1
car	1728	15	<1	<1	2±0.5	<1
expensive	1417	15	<1	<1	2±0.5	<1
cheap	2653	15	<1	<1	1±0.3	<1
monk1	124	15	1±0.2	<1	11±2.7	<1
monk3	122	15	<1	<1	4±1.3	<1
coffee	3816	15	1±0.2	<1	2±0.5	<1
banknote	1372	16	<1	<1	<1	<1
tumor	336	17	1±0.2	<1	2±0.7	<1
fico	10459	17	2±0.6	<1	4±1.2	2±0.3
tic-tac-toe	958	18	2±0.3	<1	6±1.4	<1
spect	267	22	7±2.8	<1	11±4.3	<1
bank	4521	23	3±0.8	<1	4±1.4	2±0.3
messidor	1151	24	2±0.5	<1	2±0.7	<1
hepatitis	137	34	14±5.1	<1	15±5.9	12±2.4
kr-vs-kp	3196	38	32±4.7	<1	125±25.2	4±0.6
hypothyroid	3247	39	4±1.0	<1	5±1.4	5±0.8
soybean	630	42	15±6.0	<1	22±9.0	4±0.7
anneal	812	44	3±0.7	<1	3±1.0	3±0.5
mouse	70	45	9±2.5	<1	56±21.8	1±0.2
yeast	1484	46	66±13.3	1±0.2	131±24.5	10±1.5
lymph	148	47	50±14.5	<1	73±20.5	3±0.5
vote	435	48	10±2.9	<1	10±3.0	4±0.8
HTRU2	17898	57	30±9.7	6±0.8	35±11.6	51±11.5
biodeg	1055	81	170±27.3	6±0.8	-	-
haberman	306	92	255±21.1	2±0.3	265±22.5	21±3.7
Geometric mean improvement				16.06		6.10

Table 9.4: Runtime (s) performance of methods to calculate Rashomon sets across datasets with $\lambda = 0.1$ and $n^T = 10^n$, $n \in \{1, .., 6\}$. Results are reported as mean ± standard error.

Dataset	λ	TreeFARMS	SORTD
biodeg	0.001	>3600	4
biodeg	0.01	>3600	4
biodeg	0.1	1499	9
haberman	0.001	653	2
haberman	0.01	2025	3
haberman	0.1	387	3
HTRU2	0.001	754	5
HTRU2	0.01	641	10
lymph	0.01	-	<1
lymph	0.1	-	2
vote	0.001	454	<1
vote	0.01	406	<1
vote	0.1	-	2
Geometric mean improvement			325.64

Table 9.5: Runtime (s) of methods for calculating the Rashomon set of the timeout instances in Sec. 9.B.2, with a depth budget of four, aggregated for each dataset and λ.

With the extended limit, most depth-four instances complete within the allotted time. For depth five, TreeFARMS requires substantially more memory for some larger instances and longer runtimes for others, highlighting SORTD's improvements in runtime and memory efficiency.

9.B.3. Memory Performance

Using the same experimental setup described in Section 9.B.2, we additionally evaluated the memory usage of both methods. Tables 9.7–9.9 show the detailed results (in gigabytes). Datasets for which both methods required less than 1 GB of memory are omitted from the tables, although their values are included in the geometric means of the average memory usage ratios $m_{\text{TreeFARMS}}/m_{\text{SORTD}}$ reported at the end of each table.

The geometric mean results indicate that, for each depth and λ configuration, SORTD reduces memory usage by more than an order of magnitude compared to TreeFARMS. Combined with the runtime results, these findings demonstrate that SORTD achieves superior overall efficiency in both runtime and memory performance.

Across all datasets, maximum depths, and λ values, SORTD's memory usage consistently remains below 1 GB. In contrast, TreeFARMS shows substantially higher memory requirements, particularly at greater depths for $\lambda = 0.001$ and $\lambda = 0.01$, and in datasets with larger feature spaces.

9.B.4. Rashomon Multipliers

Table 9.10 reports the minimum Rashomon multiplier ε values computed by SORTD to yield at least 10^n trees for each $n \in \{1, \ldots, 6\}$ for a subset of datasets. These results correspond to instances with $\lambda = 0.01$ and a depth budget of four. For

Dataset	λ	TreeFARMS	SORTD
anneal	0.001	570	6
anneal	0.01	513	6
biodeg	0.001	-	195
biodeg	0.01	-	104
haberman	0.001	>3600	154
haberman	0.01	-	145
haberman	0.1	-	27
hepatitis	0.001	2318	57
hepatitis	0.01	>3600	80
HTRU2	0.001	-	134
HTRU2	0.01	-	125
HTRU2	0.1	-	170
hypothyroid	0.001	1654	9
hypothyroid	0.01	-	9
kr-vs-kp	0.001	1250	12
kr-vs-kp	0.01	1323	11
kr-vs-kp	0.1	369	7
lymph	0.001	1888	9
lymph	0.01	>3600	6
lymph	0.1	-	6
mouse	0.001	1195	4
mouse	0.01	1031	5
mouse	0.1	-	3
soybean	0.001	>3600	8
soybean	0.01	2604	6
spect	0.001	1365	<1
spect	0.01	2269	<1
spect	0.1	-	2
vote	0.001	>3600	13
vote	0.01	2114	8
vote	0.1	-	9
yeast	0.001	-	16
yeast	0.01	-	13
yeast	0.1	-	21
Geometric mean improvement			197.27

Table 9.6: Runtime (s) of methods for calculating the Rashomon set of the timeout instances in Sec. 9.B.2, with a depth budget of five, aggregated for each dataset and λ. A '–' indicates results omitted due to high memory requirements.

			$d = 4$		$d = 5$	
Dataset	$\|D\|$	$\|F\|$	TreeFARMS	SORTD	TreeFARMS	SORTD
coffee	3816	15	<1	<1	1±0.0	<1
fico	10459	17	1±0.0	<1	8±0.0	<1
tic-tac-toe	958	18	<1	<1	2±0.0	<1
spect	267	22	<1	<1	3±0.0	<1
bank	4521	23	1±0.0	<1	8±0.0	<1
messidor	1151	24	<1	<1	1±0.0	<1
hepatitis	137	34	1±0.0	<1	3±0.0	<1
kr-vs-kp	3196	38	4±0.0	<1	24±0.0	<1
hypothyroid	3247	39	4±0.0	<1	24±0.0	<1
soybean	630	42	2±0.0	<1	13±0.2	<1
anneal	812	44	1±0.0	<1	9±0.0	<1
yeast	1484	46	10±0.1	<1	24±0.0	<1
lymph	148	47	1±0.0	<1	4±0.1	<1
vote	435	48	4±0.0	<1	17±0.2	<1
HTRU2	17898	57	24±0.0	<1	39±5.6	<1
biodeg	1055	81	24±0.0	<1	71±7.8	<1
haberman	306	92	1±0.0	<1	4±0.0	<1
Geometric mean improvement				24.95		44.99

Table 9.7: Memory usage (GB) of methods to calculate Rashomon sets across datasets with $\lambda = 0.001$ and $n^T = 10^n$, $n \in \{1, .., 6\}$. Results are reported as mean ± standard error. A '–' indicates results that are omitted due to their high memory requirement.

readability, values are rounded to two decimal places.

The table reveals substantial variation in the Rashomon multipliers required to achieve a given set size across datasets. For example, while $\varepsilon = 0.16$ is necessary to obtain 10 trees for the *hypothyroid* dataset, the same value yields over one million trees for the *spect* dataset. This illustrates the difficulty of selecting an appropriate Rashomon multiplier to target a desired set size. SORTD addresses this challenge through its ordered solution enumeration and anytime behavior: enumeration can be stopped once the desired number of models is reached, while still preserving the Rashomon set property—without requiring prior knowledge of the Rashomon multiplier (or bound).

9.B.5. Comparison of the Tree Ordering

A key difference between the state-of-the-art approach (TreeFARMS (Xin et al., 2022)) and SORTD lies in how trees are ordered within the Rashomon set. Fig. 9.10 illustrates this distinction. The figure reports the minimum number of trees that must be examined sequentially from the start of the Rashomon set to include all trees within the top $x\%$ of the lowest distinct objective values.

As the examples show, identifying trees within the top 20% of lowest objective values using the state-of-the-art method may require exploring nearly the entire set. In contrast, SORTD provides early access to high-quality solutions, greatly reducing

			$d=4$		$d=5$	
Dataset	$\|D\|$	$\|F\|$	TreeFARMS	SORTD	TreeFARMS	SORTD
coffee	3816	15	<1	<1	1±0.0	<1
fico	10459	17	1±0.0	<1	8±0.0	<1
tic-tac-toe	958	18	<1	<1	2±0.0	<1
spect	267	22	<1	<1	3±0.0	<1
bank	4521	23	1±0.1	<1	4±0.5	<1
messidor	1151	24	<1	<1	1±0.0	<1
hepatitis	137	34	1±0.0	<1	8±0.1	<1
kr-vs-kp	3196	38	4±0.1	<1	24±0.0	<1
hypothyroid	3247	39	2±0.4	<1	4±1.2	<1
soybean	630	42	2±0.0	<1	13±0.3	<1
anneal	812	44	1±0.0	<1	8±0.1	<1
yeast	1484	46	10±0.1	<1	24±0.0	<1
lymph	148	47	2±0.0	<1	12±0.2	<1
vote	435	48	3±0.2	<1	7±0.9	<1
HTRU2	17898	57	17±1.9	<1	22±3.0	1±0.4
biodeg	1055	81	36±4.2	<1	100±0.0	<1
haberman	306	92	2±0.2	<1	5±0.1	<1
Geometric mean improvement				21.28		30.36

Table 9.8: Memory usage (GB) of methods to calculate Rashomon sets across datasets with $\lambda = 0.01$ and $n^T = 10^n$, $n \in \{1, .., 6\}$. Results are reported as mean ± standard error. A '–' indicates results that are omitted due to their high memory requirement.

			$d=4$		$d=5$	
Dataset	$\|D\|$	$\|F\|$	TreeFARMS	SORTD	TreeFARMS	SORTD
kr-vs-kp	3196	38	3±0.4	<1	7±1.6	<1
yeast	1484	46	3±0.7	<1	6±1.6	<1
HTRU2	17898	57	2±0.7	<1	2±0.7	5±1.7
biodeg	1055	81	9±1.8	<1	-	-
haberman	306	92	1±0.1	<1	2±0.4	1±0.2
Geometric mean improvement				12.91		4.21

Table 9.9: Memory usage (GB) of methods to calculate Rashomon sets across datasets with $\lambda = 0.1$ and $n^T = 10^n$, $n \in \{1, .., 6\}$. Results are reported as mean ± standard error. A '–' indicates results that are omitted due to their high memory requirement.

	n^T					
Dataset	10^1	10^2	10^3	10^4	10^5	10^6
bank	0.08	0.13	0.16	0.23	0.25	0.32
car	0.03	0.06	0.11	0.14	0.17	0.23
hypothyroid	0.16	0.19	0.34	0.39	0.54	0.58
monk1	0.0	0.14	0.29	0.43	0.43	0.57
monk2	0.01	0.03	0.04	0.07	0.09	0.12
monk3	0.06	0.14	0.21	0.25	0.35	0.42
mouse	0.0	0.0	0.08	0.19	0.27	0.33
spect	0.0	0.02	0.05	0.07	0.1	0.13
tic-tac-toe	0.01	0.01	0.04	0.05	0.08	0.1
vote	0.16	0.17	0.32	0.32	0.44	0.47

Table 9.10: Rashomon multipliers required to obtain at least 10^n trees for instances with $\lambda = 0.01$ and depth budget four. The required multipliers vary strongly across datasets.

the overall evaluation effort.

9.B.6. Runtime Performance Without Trivial Extensions

In Section 9.B.2, we evaluated the runtime performance of SORTD while allowing trees with trivial extensions during Rashomon set computation. A trivial extension refers to a split that produces two leaf nodes with the same label (see Section 9.A.8). We now assess how excluding such trees affects SORTD's runtime performance.

Following the same experimental setup as in Section 9.B.2, we generated benchmark instances by varying the complexity penalty $\lambda \in \{0.001, 0.01, 0.1\}$ and the depth budget $d \in \{4, 5\}$. For each (dataset, λ, d) combination, SORTD was used to compute the minimum Rashomon multiplier (and corresponding Rashomon bound) required to obtain at least 10^n trees for every $n \in \{1, \ldots, 6\}$. Using these multipliers, we constructed the respective Rashomon sets to evaluate the performance of SORTD against TreeFARMS.

Runtime results are presented in Tables 9.11–9.13. For a fixed Rashomon set size, ignoring trivial extensions can increase the required multiplier values, making some instances slightly more difficult. This is reflected in a modest rise in runtime. Nevertheless, SORTD's runtime performance remains stable, consistently achieving up to two orders of magnitude speed improvement over TreeFARMS.

9.B.7. Variable Importance Analysis

Variable importance on Rashomon sets was evaluated using the *leave one feature out* procedure (Lei et al., 2018) with $\lambda = 0.01$, depth $d = 4$. For each dataset, we generated 20 stratified bootstrap samples, computed the Rashomon multiplier required to obtain at least 10,000 trees, and constructed the corresponding set using these multipliers. For each bootstrap sample, we also calculated the top-1 and

			$d = 4$		$d = 5$	
Dataset	$\|D\|$	$\|F\|$	TreeFARMS	SORTD	TreeFARMS	SORTD
breast	699	10	<1	<1	<1	<1
monk2	169	11	<1	<1	3±0.1	<1
diabetes	768	11	<1	<1	<1	<1
compas	6907	12	<1	<1	1±0.0	<1
bar-7	1913	14	<1	<1	1±0.1	<1
car	1728	15	1±0.0	<1	7±0.1	<1
expensive	1417	15	1±0.0	<1	6±0.1	<1
cheap	2653	15	1±0.0	<1	5±0.1	<1
monk1	124	15	<1	<1	6±0.8	<1
monk3	122	15	3±0.2	<1	31±3.7	<1
coffee	3816	15	1±0.0	<1	6±0.2	<1
banknote	1372	16	<1	<1	<1	<1
tumor	336	17	2±0.0	<1	18±0.1	<1
fico	10459	17	6±0.1	<1	38±0.8	2±0.1
tic-tac-toe	958	18	2±0.0	<1	22±0.3	<1
spect	267	22	29±1.2	<1	>300	<1
bank	4521	23	11±0.2	<1	121±1.7	1±0.0
messidor	1151	24	5±0.2	<1	46±1.6	<1
hepatitis	137	34	112±3.6	<1	>300	2±0.1
kr-vs-kp	3196	38	62±2.7	<1	>300	10±0.3
hypothyroid	3247	39	58±1.8	<1	>300	8±0.4
soybean	630	42	65±1.7	<1	>300	6±0.2
anneal	812	44	42±1.6	<1	>300	5±0.2
mouse	70	45	12±0.6	<1	>300	4±0.2
yeast	1484	46	240±4.2	<1	>300	16±0.4
lymph	148	47	163±8.1	<1	>300	8±0.5
vote	435	48	>300	<1	>300	9±0.1
HTRU2	17898	57	>300	4±0.1	>300	111±2.9
biodeg	1055	81	>300	4±0.1	>300	194±6.9
haberman	306	92	>300	2±0.1	>300	147±4.3
Geometric mean improvement				95.29		32.90

Table 9.11: Runtime (s) of both methods for Rashomon set computation with trivial extensions ignored, across datasets and depth budgets, for $\lambda = 0.001$ and $n^T = 10^n$, $n \in \{1, \ldots, 6\}$. Results are reported as mean ± standard error.

			$d = 4$		$d = 5$	
Dataset	$\|D\|$	$\|F\|$	TreeFARMS	SORTD	TreeFARMS	SORTD
breast	699	10	<1	<1	<1	<1
monk2	169	11	<1	<1	3±0.1	<1
diabetes	768	11	<1	<1	<1	<1
compas	6907	12	<1	<1	1±0.2	<1
bar-7	1913	14	<1	<1	1±0.1	<1
car	1728	15	1±0.1	<1	7±0.2	<1
expensive	1417	15	1±0.0	<1	7±0.1	<1
cheap	2653	15	1±0.1	<1	4±0.4	<1
monk1	124	15	2±0.2	<1	14±2.6	<1
monk3	122	15	3±0.3	<1	54±5.1	<1
coffee	3816	15	1±0.0	<1	6±0.2	<1
banknote	1372	16	<1	<1	<1	<1
tumor	336	17	2±0.1	<1	18±0.3	<1
fico	10459	17	6±0.3	<1	43±1.3	2±0.2
tic-tac-toe	958	18	2±0.0	<1	22±0.4	<1
spect	267	22	38±1.5	<1	>300	<1
bank	4521	23	10±0.6	<1	75±7.6	2±0.2
messidor	1151	24	6±0.3	<1	38±2.1	<1
hepatitis	137	34	134±3.0	<1	>300	1±0.1
kr-vs-kp	3196	38	70±3.1	<1	>300	10±0.4
hypothyroid	3247	39	37±5.3	<1	138±26.1	6±0.6
soybean	630	42	80±2.2	<1	>300	5±0.2
anneal	812	44	47±2.6	<1	>300	5±0.2
mouse	70	45	15±0.8	<1	>300	5±0.2
yeast	1484	46	234±5.6	1±0.1	>300	13±0.5
lymph	148	47	>300	<1	>300	6±0.2
vote	435	48	>300	<1	>300	7±0.5
HTRU2	17898	57	223±25.6	6±0.4	227±25.2	126±6.7
biodeg	1055	81	>300	4±0.2	>300	98±0.9
haberman	306	92	>300	3±0.2	>300	122±2.1
Geometric mean improvement				70.30		29.77

Table 9.12: Runtime (s) of both methods for Rashomon set computation with trivial extensions ignored, across datasets and depth budgets, for $\lambda = 0.01$ and $n^T = 10^n$, $n \in \{1, \ldots, 6\}$. Results are reported as mean ± standard error.

			$d = 4$		$d = 5$	
Dataset	$\|D\|$	$\|F\|$	TreeFARMS	SORTD	TreeFARMS	SORTD
breast	699	10	<1	<1	<1	<1
monk2	169	11	<1	<1	2±0.4	<1
diabetes	768	11	<1	<1	<1	<1
compas	6907	12	<1	<1	1±0.2	<1
bar-7	1913	14	<1	<1	1±0.3	<1
car	1728	15	1±0.2	<1	4±0.9	<1
expensive	1417	15	<1	<1	2±0.7	<1
cheap	2653	15	<1	<1	2±0.5	<1
monk1	124	15	2±0.3	<1	17±3.9	<1
monk3	122	15	2±0.5	<1	24±6.7	<1
coffee	3816	15	1±0.2	<1	3±0.7	<1
banknote	1372	16	<1	<1	<1	<1
tumor	336	17	1±0.2	<1	4±1.4	<1
fico	10459	17	4±0.8	<1	9±2.6	2±0.3
tic-tac-toe	958	18	2±0.3	<1	6±1.6	<1
spect	267	22	17±3.1	<1	90±25.5	<1
bank	4521	23	4±1.0	<1	8±2.4	2±0.3
messidor	1151	24	2±0.7	<1	4±1.2	1±0.2
hepatitis	137	34	34±10.0	<1	41±13.0	2±0.3
kr-vs-kp	3196	38	53±7.9	<1	178±28.2	5±0.8
hypothyroid	3247	39	21±6.8	<1	38±13.7	6±0.9
soybean	630	42	31±8.9	<1	42±12.1	5±0.8
anneal	812	44	11±3.4	<1	16±5.3	3±0.5
mouse	70	45	21±6.3	<1	110±28.2	1±0.3
yeast	1484	46	114±16.0	1±0.2	185±24.7	10±1.5
lymph	148	47	112±25.0	<1	122±26.8	4±0.7
vote	435	48	64±24.1	<1	67±24.7	5±1.1
HTRU2	17898	57	49±13.5	5±0.7	70±22.0	50±9.2
biodeg	1055	81	177±26.9	6±0.8	-	-
haberman	306	92	259±21.4	2±0.3	276±23.1	22±3.7
Geometric mean improvement				28.30		11.71

Table 9.13: Runtime (s) of both methods for Rashomon set computation with trivial extensions ignored, across datasets and depth budgets, for $\lambda = 0.1$ and $n^T = 10^n$, $n \in \{1, \ldots, 6\}$. Results are reported as mean ± standard error. A "–" indicates runs omitted due to excessive memory usage.

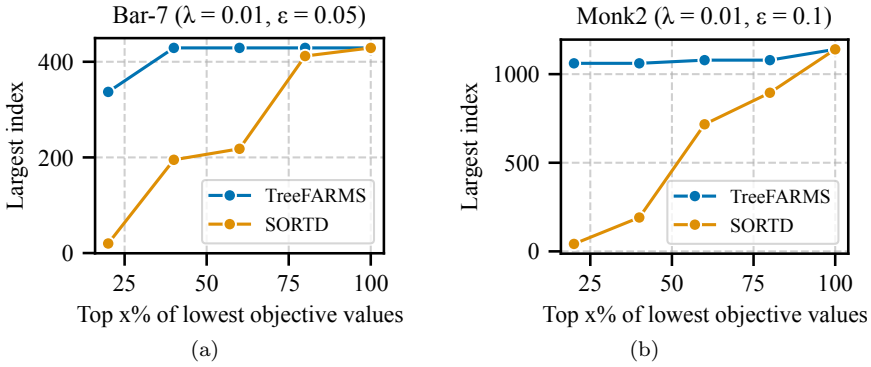


Figure 9.10: Largest index of the trees that belong to the top-x% of the lowest distinct objective values.

top-100 Rashomon sets and ranked features according to the increase in area under the tree-index versus objective-value curve when each feature was removed. Kendall's τ and Jaccard index were used to evaluate the stability of the full and top-5 variable ranking, respectively. Table 9.14 presents the results.

The stability of the top-1 tree versus the top-10,000 trees is low for most datasets with only five of them satisfying $\tau \geq 0.7$, and 19 of them have a Jaccard index of value at least 0.8. In contrast, for the top-100 versus top-10,000 trees, 17 datasets satisfy $\tau \geq 0.7$ and 26 exceed a Jaccard index exceeding 0.8, indicating that smaller Rashomon sets can yield similar variable importance values as larger Rashomon sets.

9.B.8. Regression Results

This section provides the details on the regression experiment shown in Fig. 9.7 and additionally analyses the runtime performance of SORTD for regression.

Data For the regression experiments, we use the datasets from Zhang et al. (2023) and also follow their binarization of non-binary features.[3] The datasets can also be obtained from the UCI Machine Learning repository (Dua and Graff, 2017). For all datasets, we normalize the regression label by subtracting the mean and dividing by the standard deviation.

Comparison with CART and STreeD Since there are, to the best of our knowledge, no previous methods to enumerate the whole Rashomon set of regression trees, we follow the set-up by Xin et al. (2022), who did a similar experiment for classification trees: we compare SORTD with both the heuristic CART (Breiman et al., 1984) and the optimal method STreeD (Van der Linden et al., 2023) when those are called repeatedly on random samples of the data. STreeD is a state-of-the-art optimal method for computing optimal regression trees (Van den Bos et al., 2024).[4]

[3] https://github.com/ruizhang1996/regression-tree-benchmark
[4] https://github.com/algtudelft/pystreed

	Kendall's τ		Jaccard Index	
Dataset	$n^T = 1$	$n^T = 100$	$n^T = 1$	$n^T = 100$
diabetes	0.71	0.89	1.0	1.0
anneal	0.02	0.59	0.8	1.0
bank	-0.22	0.64	0.4	1.0
banknote	0.68	0.93	1.0	1.0
bar-7	0.56	0.78	1.0	1.0
breast	0.69	0.91	0.6	1.0
car	0.70	0.83	0.8	0.8
cheap	0.47	0.83	0.8	1.0
coffee	0.73	0.81	1.0	1.0
compas	0.42	0.88	0.6	0.8
expensive	0.79	0.79	0.8	0.6
fico	0.13	0.79	0.4	0.8
haberman	0.25	0.46	0.4	0.8
hepatitis	0.43	0.62	0.2	0.4
HTRU2	0.12	0.21	0.6	1.0
hypothyroid	0.11	0.42	0.4	0.8
kr-vs-kp	0.39	0.49	0.8	0.8
lymph	0.50	0.72	0.8	0.8
messidor	0.60	0.80	0.8	1.0
monk1	0.66	0.66	0.6	0.8
monk2	0.75	0.75	1.0	0.8
monk3	0.68	0.64	1.0	1.0
mouse	0.34	0.51	1.0	0.8
tumor	0.57	0.85	0.8	1.0
soybean	0.26	0.79	0.8	1.0
spect	0.58	0.74	0.8	0.6
tic-tac-toe	0.61	0.67	1.0	1.0
vote	0.06	0.69	0.2	0.8
yeast	0.38	0.74	0.8	0.8

Table 9.14: Variable importance similarity compared to LOFO based on $n^T = 10^4$.

In this experiment, we impose a 60 seconds timeout for each method. CART and STreeD are run repeatedly within that time budget on random samples of 50% of the total dataset. All resulting unique trees are kept. SORTD, on the other hand, is run once and collects trees in the Rashomon set in order until time-out or until the Rashomon set is exhausted. We use the Rashomon multiplier $\varepsilon = 0.1$, complexity cost $\lambda = 0.001$, and maximum depth $d = 4$.

Fig. 9.7 in the main text shows how SORTD can find orders of magnitude more trees in the Rashomon set than either CART or STreeD within the time limit. For the *Synchronous Machine Dataset*, SORTD finds the whole Rashomon set within two seconds, whereas both STreeD and CART find only a fraction in the allotted 60 seconds. For both *Airfoil Self-Noise* and *Seoul Bike Sharing Demand*, SORTD finds orders of magnitude more trees than either CART or STreeD in the given time limit. These results confirm what Xin et al. (2022) also concluded for classification: enumerating the whole Rashomon set can be done best with a dedicated method.

Runtime Additionally, we analyse the runtime performance of SORTD to calculate the Rashomon set of regression trees. Here, we set the complexity cost $\lambda = 0.01$, the Rashomon multiplier to $\varepsilon = 1.0$, the maximum number of trees $n^T = 10^6$, and we test both with a maximum depth of four and five. We run all experiments five times on all datasets with a time-out of 300 seconds.

Dataset	$\|D\|$	$\|F\|$	$d = 4$	$d = 5$
Airfoil Self-Noise	1503	17	2 ± 0.1	2 ± 0.1
Air Quality	111	16	2 ± 0.3	9 ± 0.9
Energy Efficiency (Cooling)	768	27	80 ± 5.0	> 300
Energy Efficiency (Heating)	768	27	94 ± 3.4	> 300
Household	2049280	15	53 ± 0.1	134 ± 1.0
Medical Cost Personal	1338	16	2 ± 0.0	12 ± 0.1
Optical Interconnection Network	640	29	< 1	13 ± 0.0
Real Estate Valuation	414	18	13 ± 0.0	18 ± 0.1
Seoul Bike Sharing Demand	8760	32	> 300	> 300
Servo	167	15	2 ± 0.1	4 ± 0.2
Synchronous Machine	557	12	< 1	< 1
Yacht Hydrodynamics	308	35	14 ± 0.5	157 ± 4.9

Table 9.15: SORTD runtime (s) performance of methods to calculate Rashomon sets for regression trees across datasets with $\lambda = 0.01$ and $n^T = 10^6$. Results are reported as mean ± standard error.

Table 9.15 shows that SORTD successfully enumerates the top one million trees for all benchmark datasets within the time limit for $d = 4$, except for one, and for all but three for $d = 5$.

In comparison to the experiments on classification trees shown above in Appendix 9.B.2, the regression tree Rashomon set is more time intensive to compute. We think this is because the regression loss allows for many more unique loss values.

Since SORTD combines solutions with the same value (see Appendix 9.A.2), more unique loss values result in a higher runtime. Runtime performance can likely be improved by setting a higher tolerance for which solution values are considered the 'same'. Currently, SORTD uses a tolerance of 10^{-4}.

9.B.9. Equality-of-Opportunity Results

This section provides the details on the equality-of-opportunity experiment shown in Fig. 9.8 and additionally analyses the runtime performance of SORTD for evaluating such a second objective.

Data For the equality-of-opportunity experiments, we use the datasets from Le Quy et al. (2022) and follow their suggested binarization of non-binary features. In Table 9.16, we report which feature is selected as the discrimination sensitive feature. References to the original datasets can be found here (Dutch Central Bureau for Statistics, 2001b; Cortez and Silva, 2008; Moro et al., 2014; Strack et al., 2014; Angwin et al., 2016b; Dua and Graff, 2017; Kuzilek et al., 2017).

Pareto front Fig. 9.8 shows how SORTD can be used to generate a Rashomon set for sparse classification trees, which are then evaluated using a secondary objective: equality of opportunity. Since SORTD generates the solutions in increasing order of its objective, it examines the most accurate trees first. Therefore, when halting the search at any time, a (partial) Pareto front over the primary and secondary objectives can be computed. The results in Fig. 9.8 show that among the top 10^7 sparse depth-four classification trees for the *Adult* dataset, all trees have at least 2% discrimination, i.e., the true positive rate of one group is at least 2% higher than another. For the *Bank Marketing* dataset, a tree with zero discrimination is found among the top 10^7 trees. For the *Compas* dataset the most fair dataset in the top 10^7 trees still has 2.5% discrimination. It also shows that by reducing accuracy by only 0.5%, the discrimination score can be lowered from 7% to 2.5%.

Runtime comparison We evaluate SORTD's efficiency in evaluating a second objective such as equality of opportunity by comparing it with the optimal dynamic programming approach STreeD (Van der Linden et al., 2023), which supports optimizing accuracy and equality of opportunity. We do not compare with the mixed-integer programming approach by Jo et al. (2023) since Van der Linden et al. (2023) report runtimes several orders of magnitude lower than theirs, while computing the same optimal solutions.

We run SORTD by iteratively generating the next best n^T trees according to the regularized accuracy objective. We then evaluate the newly generated trees as described in Appendix 9.A.9 using the equality-of-opportunity metric. If a tree is found within the pre-set discrimination limit δ, we stop. Otherwise, SORTD generates the next batch of n^T trees and repeats. Since SORTD yields trees in order of the regularized accuracy objective, the first tree it finds within the discrimination limit must be optimal with respect to the regularized accuracy objective.

In the comparison with STreeD, there are a couple of small differences that may influence runtime: (1) We run SORTD using the regularized accuracy objective,

so each leaf node is penalized with the sparsity penalty λ. In this experiment, we set $\lambda = 0.01$. STreeD's equality-of-opportunity optimization task by default does not consider such regularization. (2) STreeD also considers non-majority labels in leaf nodes, whereas SORTD by default does not. Because of these two differences, STreeD and SORTD are not guaranteed to find the same optimal solution, although we verified that the solutions are close. Since our main aim in this experiment is to validate that SORTD can successfully be used to post-evaluate a secondary objective, we consider these differences acceptable.

Dataset	Sensitive feature	$\|D\|$	$\|F\|$	STreeD	SORTD
Adult	Gender	45222	18	**1 ± 0.0**	**1 ± 0.0**
Bank marketing	Married	45211	47	8 ± 0.1	**7 ± 0.0**
Communities & crime	Race	1994	98	**6 ± 0.0**	21 ± 0.1
Compas recid.	Race	6172	10	**<1**	**<1**
Compas viol. recid.	Race	4020	10	**<1**	1 ± 0.0
Diabetes	Race	45715	129	**36 ± 0.0**	58 ± 0.2
Dutch census	Gender	60420	59	**9 ± 0.0**	79 ± 0.5
German credit	Gender	1000	70	32 ± 0.2	**3 ± 0.0**
KDD census income	Race	284556	118	**27 ± 0.1**	134 ± 0.6
Lawschool	Race	20798	20	**<1**	4 ± 0.0
OULAD	Gender	21562	46	17 ± 0.2	**7 ± 0.0**
Ricci	Race	118	5	**<1**	**<1**
Student Portuguese	Gender	649	56	**1 ± 0.0**	2 ± 0.0
Student mathematics	Gender	395	56	**<1**	6 ± 0.0

Table 9.16: Runtime (s) performance to compute an optimal fair tree with at most 1% discrimination, and maximum depth $d = 3$, averaged over five runs. For SORTD, we set $\lambda = 0.01$ and iteratively evaluate 10^5 trees. Results are reported as mean ± standard error. Best results are bold.

Table 9.16 shows the mean runtime performance of STreeD and SORTD to find fair and optimal trees with max-depth $d = 3$, discrimination limit $\delta = 1\%$, and sparsity penalty $\lambda = 0.01$ (for SORTD). With SORTD, we iteratively produce and evaluate $n^T = 10^5$ trees until a tree within the discrimination limit is found, or until the time-out of 300 seconds. The results show that SORTD remains close in performance to STreeD, and for some datasets, such as *German credit* and *OULAD*, even performs significantly better. These results are obtained by evaluating the secondary objective using callbacks to Python. Hence, if runtime performance was the main concern, these results could be further improved by computing this also in C++.

Concluding, these results show that using Rashomon sets to optimize one (totally ordered) objective, and later evaluating the Rashomon set (possibly iteratively) using a second objective (such that this multi-objective optimization task is only partially ordered) is promising.

10

Conclusion

In this thesis, we have advanced the scalability, applicability, and understanding of optimal decision trees (ODTs), thus contributing to the adoption of inherently interpretable models for machine learning. Throughout the thesis, we have extended the applicability of dynamic programming (DP) for computing ODTs to a wide variety of learning tasks, such as regression, survival analysis, prescriptive policy generation, and handling group fairness constraints, while also improving scalability by several orders of magnitude over the state of the art. We provided an in-depth empirical comparison between greedy and optimal learning methods to refute (and also confirm) several claims in the literature. Last, we also significantly improved scalability to optimize ODTs directly for numerical features and for computing the whole Rashomon set in order.

In the sections below, we highlight the main conclusions of this thesis by answering the research questions, reiterating the contrast between model- and search-based ODT approaches, and considering limitations and future work.

10.1. Answers to the Research Questions

This dissertation addressed four research questions about learning optimal decision trees: (i) supporting constraints while maintaining scalability, (ii) improving our understanding of the difference between greedy and optimal learning approaches, (iii) efficiently handling numerical features, and (iv) efficiently enumerating all decision trees in the Rashomon set. We discuss each of the four research questions below.

RQ1. Under what conditions can dynamic programming methods for optimal decision trees be generalized to machine learning tasks beyond classification?

In the first research question, we investigated how to generalize the search-based approach to optimal decision trees to other learning tasks beyond classification.

Necessary and sufficient conditions for DP Dynamic programming requires problems to be *separable*. Therefore, previous general DP approaches required solution values to be *additive* (Nijssen and Fromont, 2010; Lin et al., 2020), but

we showed that this condition is not necessary. Instead, we defined a new property *order preservation*, which can informally be described as follows: any combination of two optimal solutions to subproblems should always dominate a combination that contains at least one sub-optimal subproblem solution.

To the best of our knowledge, our new condition of order preservation also generalizes DP theory beyond decision tree learning. While it is, for example, similar to the notion of monotonicity presented by Karp and Held (1967), we do not assume solution values to be real-valued, and in contrast to Elmaghraby (1970), we do not assume that solution values are totally ordered.

We showed that this new property order preservation, together with the *anti-monotonicity* of constraints (Nijssen and Fromont, 2010) and the Markov property on the transition and cost function form the necessary and sufficient conditions for using DP for learning optimal decision trees (Chapter 4). These conditions help inform which objectives and constraints can be solved best with DP and we used it to provide more scalable alternatives for model-based approaches for a variety of learning tasks.

Algorithmic improvements To improve scalability, we also explored the conditions for and applicability of (i) branch-and-bound, (ii) similarity lower bounding, and (iii) preprocessing counts for an improved depth-two subroutine (Chapter 4). We provided a new technique for computing subproblem upper bounds for multi-dimensional additive objectives, and showed that both the similarity lower bound and depth-two preprocessing technique require an *element-wise additive* breakdown of the cost function. We contributed *survival analysis* (Chapter 5) and *(simple linear) regression* (Chapter 6) as specific examples where such a breakdown can be defined. Similarly, we explored how these same techniques can be applied to optimizing decision trees under a *group fairness* constraint (Chapters 3 and 4). Together, these algorithmic improvements have significantly improved scalability across a variety of learning tasks, as we detail below.

Performance Compared to MIP baselines, our approaches improve scalability by several orders of magnitude for group fairness constraints (Aghaei et al., 2019; Jo et al., 2023), prescriptive policy generation (Jo et al., 2021), and regression (Verwer and Zhang, 2017; Dunn, 2018). This result aligns with the same performance difference already observed between DP and MIP when maximizing classification accuracy (Lin et al., 2020; Demirović et al., 2022). The reason for such a large performance difference is that our DP approach can exploit the separability property: we solve subtrees independently, whereas MIP is not able to exploit this.

In comparison to previous DP approaches, we also see a significant improvement for regression compared with Zhang et al. (2023) and for nonlinear classification metrics compared with Demirović and Stuckey (2021). These performance gains are the result of incorporating and adapting the depth-two subroutine by Demirović et al. (2022) to regression, and improving the computation of upper and lower bounds for the nonlinear classification metrics.

Under a limiting fixed depth limit, we also repeatedly see significant out-of-sample performance improvements compared to the greedy-top-down induction. For

regression, this difference was not so pronounced, but here the depth limit was rather permissive. In follow-up work, we showed that this is expected as the accuracy for the greedy and optimal approach converge for larger size limits (Chapter 7). We also repeatedly see that an optimal model can achieve the same performance as a greedy model while using fewer nodes. In comparison with the coordinate descent method by Dunn (2018), the differences are smaller.

RQ2. What are the differences, strengths, and weaknesses of optimal and greedy decision tree learning approaches?

In the second research question, we further investigated how top-down greedy induction decision tree learners compare with an optimal learning approach (Chapter 7).

Effect of the optimization objective and hyperparameter tuning We showed that optimal decision tree methods perform worse when trained with *strictly concave* objectives, while such objectives are required for greedy top-down induction approaches (Kearns and Mansour, 1996). The results also indicate that an objective that includes a regularization component for noise can be an effective method to combat overfitting, in addition to the normal regularization.

The differences in accuracy between most hyperparameter tuning approaches for ODTs are small, but the tuning methods may yield differently sized trees. If runtime is a concern, tuning only the depth is efficient, but may return trees that are larger than necessary. Still, compared to no hyperparameter tuning at all, most hyperparameter tuning methods significantly improve out-of-sample accuracy and reduce model size.

Comparing optimal and greedy decision trees We reviewed previous comparisons between greedy and optimal decision three methods, several of which had mutually contradicting results. We observed (i) that many of the comparisons were limited due to scalability issues; (ii) that some compared greedy and optimal methods under the same depth limit, while others imposed a depth limit on the optimal method, but not on the greedy method, resulting in conflicting results; and (iii) that in many comparisons the hyperparameters of one or more methods were improperly or not tuned at all. To prevent this in future comparisons, we proposed a set of best practices for comparing with optimal decision tree methods.

We conducted a new comparison according to these best practices and applied it to test six claims in the literature, three of which we highlight here. We confirm the most important claim that optimal decision trees obtain a better *accuracy-interpretability trade-off* than their greedy counterparts. To establish this claim, we developed a new metric that captures both aspects: size-weighted accuracy. Confirming this claim establishes the original motivation to research ODTs: to obtain ML models that are both accurate and interpretable.

In our experiments, we found no evidence for the claim that ODTs are more prone to overfitting than greedy decision trees (Dietterich, 1995). The original claim compared properly regularized greedy trees with unregularized optimal decision trees. If, however, both methods are subject to proper hyperparameter tuning, we see in fact the opposite: ODTs seem *less* sensitive to noise.

Similarly, we found no evidence for the claim that the differences between optimal and greedy trees diminish with more data. Costa and Pedreira (2023) based this claim on the experiments by Murthy and Salzberg (1995). Murthy and Salzberg, however, do not directly test this claim, but only observe as a side remark that the greedy heuristic gets more accurate with more (noise-free) training data. This statement we can confirm. However, we also observe that depth-constrained greedy trees remain significantly less accurate than ODTs even with increasing data, whereas unconstrained greedy trees are much larger than ODTs, and increasingly so for more data. Therefore, the results are such to raise the hypothesis that the differences in fact get larger with more data.

RQ3. What algorithmic techniques can be exploited to improve the scalability of learning optimal decision trees directly on numerical feature data?

In the third research question, we explored algorithmic improvements for dealing with numerical feature data (Chapter 8).

We introduced three new *pruning techniques* based on the similarity lower bound proposed by Hu et al. (2019) and Demirović et al. (2022) (Chapter 8). These pruning techniques exploit the fact that if we have computed an optimal subtree given a specific split threshold for the root node of a (sub)tree, we can immediately infer bounds on the solution of all splits with a threshold that is only slightly different. By applying these pruning techniques we are able to prune large portions of the search space.

Additionally, we exploited the sorted property of numerical data to enable *incrementally* computing all possible depth-two trees given a fixed root split. This incremental subroutine avoids having the expensive step of splitting the dataset and computing the score for each possible split, and thus results in a major runtime reduction.

These improvements together result in one to two orders of magnitude runtime improvement over the state-of-the-art method Quant-BnB (Mazumder et al., 2022) and even more in comparison to MIP (Bertsimas and Dunn, 2017) and SAT baselines (Shati et al., 2023b).

RQ4. What algorithmic techniques can be used to improve the scalability of enumerating the Rashomon set of decision trees in order?

For the fourth research question, we explored how to efficiently enumerate all close-to-optimal models in order (Chapter 9).

We developed an algorithm that improves existing DP techniques to compute the whole Rashomon set up to two orders of magnitude faster than the state of the art while generating each model in that set *in-order* of its objective value. Specifically, we adapted the depth-two subroutine by Demirović et al. (2022) to efficiently compute all partial solutions that may potentially appear as submodels in the Rashomon set. We obtained in-order computation by introducing a novel method to compute the cartesian sum in order, evaluating subproblems as lazily as possible.

As before, our approach is not limited to classification, but can be used for any *separable* and *totally ordered* objective. For example, we applied our method without

any significant effort to obtain the Rashomon set of regression trees.

In addition, any separable objective can be used to post-hoc evaluate the Rashomon set. We provided an example for this by obtaining the Rashomon set optimized by regularized accuracy, and post-hoc evaluating the set using equality-of-opportunity as a metric next to accuracy.

10.2. Model- and Search-Based Approaches

In the last couple of years, a large number of papers have proposed either *model*-(MIP, SAT, CO, CP) or *search*-based (DP, BnB) approaches for ODTs (Chapter 2), but qualitative and quantitative comparisons between the two paradigms were limited. This has left the question which approach is better and when mostly unanswered. Therefore, in this concluding section, we revisit the advantages and disadvantages of the two, focusing on (i) global constraints and objectives, (ii) scalability, and (iii) handling numerical features.

Global objectives and constraints In this thesis, we have precisely defined the "separability" condition for the use of DP for ODTs. This condition reveals both the breadth of DP's applicability (as also evidenced by this thesis' application of DP across a wide variety of objectives and constraints) and its limitation. It reveals that certain learning tasks are hard to optimize with DP, e.g., adversarial robustness (Vos and Verwer, 2022).

Similarly, model-based approaches have their own limitations and conditions. For mixed-integer linear programming, the condition is linear constraints and objectives. For example, F1-score and the survival analysis loss function used in Chapter 5 are non-linear loss functions and cannot trivially be solved with MIP. Similarly, SAT requires objectives or constraints that can easily be expressed as Boolean formulas. Therefore, each of these solution methods has its own limitations in what objectives and constraints are easy to model.

Scalability Throughout this thesis we confirmed that the scalability of search-based approaches is better than the model-based approaches, provided the separability conditions holds. The search-based approaches can exploit the separability property by solving subtrees independently, whereas the model-based approaches (except for the CP approach) cannot. We compared our DP approach to several MIP models and repeatedly observed several orders of magnitude better scalability for DP: for handling group fairness constraints, prescriptive policy models, a variety of regression learning tasks, and for classification (Chapters 3, 4, 6 and 8).

Numerical features At the start of this thesis, Mazumder et al. (2022) provided the only search-based approach that could handle numerical features well. With the algorithmic improvements that we introduced, we can now solve problems that take multiple hours with MIP and SAT approaches in less than a second. We even observed cases where the LP relaxation of a MIP model was unsolved after four hours, whereas our DP solution was done within a second. These results highlight the fact that, also for numerical feature data, search-based approaches now significantly outperform model-based solvers in scalability (Chapter 8).

Conclusion With the advancements presented in this thesis, the two drawbacks of search-based approaches have been addressed (support for a variety of global objectives and constraints and handling numerical features). Therefore, given the large difference in scalability, we can now confidently recommend future researchers and users to investigate the possibility of using DP for computing ODTs first, and only if this is not feasible, to fall back to model-based approaches.

In short, if a learning task is not separable, one can use model-based approaches. But if it is separable, search-based approaches are clearly the better choice. The availability of easy-to-adapt frameworks such as DL8.5 (Aglin et al., 2020a,b) and STreeD (Van der Linden et al., 2023) also make this possible.

10.3. Limitations and Future Work

Naturally, the work done has its limitations and assumptions. We discuss below the possible impact and how to extend the work to address these.

Binarization While Chapter 8 shows how to significantly improve scalability for handling numerical feature data, all other chapters in this thesis assume all feature data is binarized beforehand. This is a common assumption among both search-based (e.g., Lin et al., 2020; Demirović et al., 2022) and model-based approaches (e.g., Narodytska et al., 2018; Aghaei et al., 2024). Future work could improve and extend the methods and experiments presented in these chapters to also handle numerical feature data directly.

However, while *additivity* is not a necessary condition for the use of DP (Chapter 4), our three pruning techniques for handling numerical features efficiently do require additive objectives. Two of the pruning techniques depend on the similarity lower bound and thus also require the stronger condition of a *per-instance element-wise additive* breakdown of the objective. For example, our formulation of the group fairness constraint and nonlinear classification metrics are not additive, and our pruning approaches cannot trivially adapt to these learning tasks. Therefore, our method for optimizing with numerical features is less general than the conditions for DP specified in Chapter 4.

Nonetheless, most of the learning tasks considered in this thesis also satisfy additivity and future work could therefore easily adapt them to handle numeric features (e.g., regression, survival analysis, prescriptive policy generation, and cost-sensitive classification). We leave the other learning tasks as future work.

Binary axis-aligned trees Another limitation of all methods proposed in this thesis is that they all assume binary trees and all splits are assumed to be axis-aligned. Specifically for categorical features, the introduction of multi-way splits might improve model sparsity and/or interpretability. The limitation of axis-aligned splits is motivated by interpretability, since oblique (or multivariate) splits are typically considered hard to comprehend for human users. Multi-way splits could be investigated in future work.

Search strategies and anytime performance All methods proposed in this thesis use depth-first search and aim to minimize the time to prove optimal solutions.

As introduced in the background section, there are alternative *search strategies*, such as best-first and AND-OR search. Recent work also explores search strategies that improve anytime performance, such as limited discrepancy search (Kiossou et al., 2022), a variant of depth-first search that changes the node visit order (Demirović et al., 2023), anytime beam search (Kiossou et al., 2025), and an anytime adaptation of our approach proposed in Chapter 8 (Kiossou et al., 2026). Such anytime approaches are more promising for larger datasets and size budgets where finding and proving the optimal solution within a reasonable time limit is unlikely. However, while the literature shows this variety of search strategies, it provides no in-depth comparison of these strategies and therefore our understanding of what method is best and when, is limited. Therefore, future work should investigate the advantages and disadvantages of each of these search strategies.

In addition, current anytime approaches only consider anytime performance on the training data, but how to obtain good anytime out-of-sample performance is left unexplored. Cross validation as an outer loop around an anytime algorithm defeats the purpose of anytime search. The Rashomon set could provide an interesting new perspective since all models that would result from cross validation most likely also appear in the Rashomon set of the whole training dataset. As far as I know, it is not investigated yet if the Rashomon set can provide a new way of cross validation without requiring an outer loop and thus provide anytime out-of-sample performance.

Small representative Rashomon sets In Chapter 9, we improved the scalability of finding the whole Rashomon set of decision trees. However, our current approach has several directions for future work. First and similarly as above, our approach (and also the previous approach) assumes numerical and nominal features have been coarsely binarized to remain scalable. Second, Rashomon sets are used to obtain better feature importance analysis, but feature importance attribution is harder when features are already binarized. Third, not only is the search space of decision trees huge, but for Rashomon sets the output space can be too large to enumerate as well. Even for a small dataset, depth limit, and coarse binarization, we observed Rashomon sets of over 10^{12} models. Without a coarse binarization, these sets will be even larger. So, while we want to move away from coarse binarizations, it is not clear how to do this with current approaches. Fourth, the returned Rashomon set includes all unique models within a performance bound, even if they differ only by the slightest. As observed, this number can be huge, and such huge sets are too large for domain experts to interpret. Fig. 10.1 highlights some of these drawbacks.

This raises the question of how to generate smaller sparser subsets of the Rashomon set that provide a good representation of the complete set, to enable interpretation by domain experts, and speed up down-stream analysis. Similar problems have been encountered in frequent pattern mining, e.g., requesting all frequent patterns returns a set that is too large to parse. Instead, pattern set mining searches for a small set of representative patterns, typically orders of magnitude smaller than the original set (e.g., Vreeken et al., 2011). The similarities with the Rashomon set are striking, but as far as I know, not explored yet.

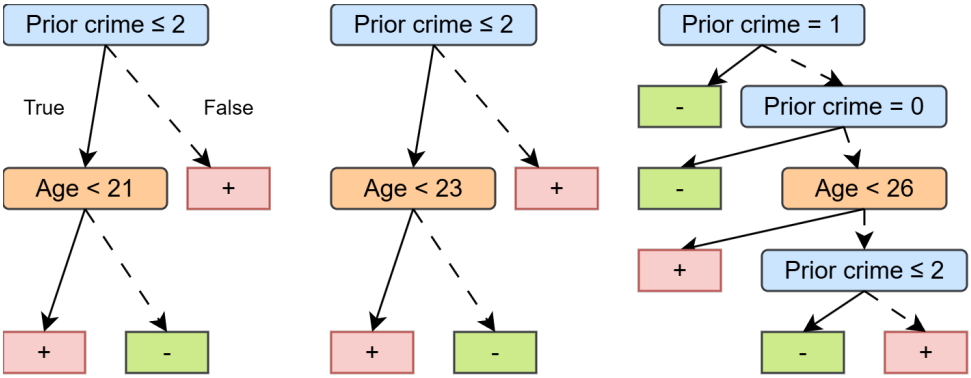


Figure 10.1: Three trees in the Rashomon set of the COMPAS dataset (Angwin et al., 2016a) to predict high (+) and low (-) recidivism risk for supporting parole decisions. The trees obtain similar training accuracy: 65.4%, 65.1%, and 66.1%, respectively. The first two trees are almost identical and motivate to filter the Rashomon set on basis of diversity. The last tree highlights the effect of suboptimal binarization because replacing the first two branching nodes with Prior crime ≤ 1 reduces the tree size.

Decision diagrams Finally, recent work investigated the use of *decision diagrams* as an alternative to decision trees. Unlike decision trees that only have *split* nodes, decision diagrams also allow paths to *merge* back together. Such diagrams might be able to solve two weaknesses of decision trees: *fragmentation* and *replication*. Fragmentation occurs in decision trees because after each split, the nodes in the next layer have less data to base their decisions on. Replication means that similar patterns could reoccur in separate branches of the tree. Recent work explores the use of MIP (Florio et al., 2023) and SAT approaches (Hu et al., 2022; Shati et al., 2023a; Cabodi et al., 2024) for computing optimal classification diagrams. However, as with decision trees, the scalability of these model-based approaches is limited. Unlike decision trees, on the other hand, it is not easy to construct a search-based approach for optimizing decision diagrams for classification, since the separability property does not hold for diagrams as it does with decision trees.

Future prospect The contributions in this thesis have significantly improved scalability of finding ODTs, often by several orders of magnitude, have broadened the applicability of DP to a breadth of learning tasks, deepened our understanding of the differences between greedy and optimal approaches, and extended training ODTs to efficiently return all close-to-optimal models. As a result, this thesis has advanced the use of optimal decision trees as machine learning models that are both accurate and interpretable.

Bibliography

Aalen, Odd (1978). Nonparametric Inference for a Family of Counting Processes. *The Annals of Statistics* 6.4, pp. 701–726.

Agarwal, Alekh, Alina Beygelzimer, Miroslav Dudík, John Langford, and Hanna Wallach (2018). "A reductions approach to fair classification." In *Proceedings of ICML-18*.

Aghaei, Sina, Mohammad Javad Azizi, and Phebe Vayanos (2019). "Learning Optimal and Fair Decision Trees for Non-Discriminative Decision-Making." In *Proceedings of AAAI-19*, pp. 1418–1426.

Aghaei, Sina, Andrés Gómez, and Phebe Vayanos (2024). Strong Optimal Classification Trees. *Operations Research* 73.4, pp. 2223–2241.

Aglin, Gaël, Siegfried Nijssen, and Pierre Schaus (2020a). "Learning Optimal Decision Trees Using Caching Branch-and-Bound Search." In *Proceedings of AAAI-20*, pp. 3146–3153.

Aglin, Gaël, Siegfried Nijssen, and Pierre Schaus (2020b). "PyDL8.5: a Library for Learning Optimal Decision Trees." In *Proceedings of IJCAI-20*, pp. 5222–5224.

Aglin, Gaël, Siegfried Nijssen, and Pierre Schaus (2022). "Learning Optimal Decision Trees Under Memory Constraints." In *Proceedings of ECML-PKDD-22*.

Akay, Mehmet (2018). *Optical Interconnection Network*. UCI Machine Learning Repository.

Aldinucci, Tommaso and Matteo Lapucci (2024). Loss-Optimal Classification Trees: A Generalized Framework and the Logistic Case. *Transactions in Operations Research* 32.2, pp. 323–350.

Alès, Zacharie, Valentine Huré, and Amélie Lambert (2024). New optimization models for optimal classification trees. *Computers & Operations Research* 164, p. 106515.

Alòs, Josep, Carlos Ansótegui, and Eduard Torres (2023). Interpretable decision trees through MaxSAT. *Artificial Intelligence Review* 56.8, pp. 8303–8323.

Amram, Maxime, Jack Dunn, and Ying Daisy Zhuo (2022). Optimal Policy Trees. *Machine Learning* 111.7, pp. 2741–2768.

Andersen, Hayden, Andrew Lensen, Will Browne, and Yi Mei (2023). "Producing Diverse Rashomon Sets of Counterfactual Explanations with Niching Particle Swarm Optimization Algorithms." In *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 393–401.

Angwin, Julia, Jeff Larson, Surya Mattu, and Lauren Kirchner (May 2016a). *Machine Bias*. URL: `https://www.propublica.org/article/machine-bias-risk-assessments-in-criminal-sentencing`.

Angwin, Julia, Jeff Larson, Surya Mattu, and Lauren Kirchner (May 2016b). *Machine Bias*. URL: https://www.propublica.org/article/machine-bias-risk-assessments-in-criminal-sentencing.

Arslan, Elif, Jacobus G. M. van der Linden, Serge Hoogendoorn, Marco Rinaldi, and Emir Demirović (2025). "SORTeD Rashomon Sets of Sparse Decision Trees: Anytime Enumeration." In *Advances in NeurIPS-25*.

Assis, André, Jamilson Dantas, and Ermeson Andrade (2025). The performance-interpretability trade-off: A comparative study of machine learning models. *Journal of Reliable Intelligent Environments* 11.1, p. 1.

Athey, Susan and Guido Imbens (2016). Recursive partitioning for heterogeneous causal effects. *Proceedings of the National Academy of Sciences* 113.27, pp. 7353–7360.

Avellaneda, Florent (2020). "Efficient Inference of Optimal Decision Trees." In *Proceedings of AAAI-20*, pp. 3195–3202.

Avellaneda, Florent (2025). "Learning Optimal Oblique Decision Trees with (Max)SAT." In *Proceedings of IJCAI-25*.

Azizi, Mohammad Javad, Phebe Vayanos, Bryan Wilder, Eric Rice, and Milind Tambe (2018). "Designing Fair, Efficient, and Interpretable Policies for Prioritizing Homeless Youth for Housing Resources." In *Proceedings of CPAIOR-18*.

Bellman, Richard (1957). *Dynamic Programming*. Princeton, NJ: Princeton University Press.

Berk, Richard, Hoda Heidari, Shahin Jabbari, Michael Kearns, and Aaron Roth (2021). Fairness in criminal justice risk assessments: The state of the art. *Sociological Methods & Research* 50.1, pp. 3–44.

Bertsimas, Dimitris and Jack Dunn (2017). Optimal classification trees. *Machine Learning* 106.7, pp. 1039–1082.

Bertsimas, Dimitris and Jack Dunn (2019). *Machine Learning Under a Modern Optimization Lens*. Belmont, MA: Dynamic Ideas.

Bertsimas, Dimitris, Jack Dunn, Emma Gibson, and Agni Orfanoudaki (2022). Optimal Survival Trees. *Machine Learning* 111.8, pp. 2951–3023.

Bertsimas, Dimitris, Jack Dunn, and Nishanth Mundru (2019). Optimal Prescriptive Trees. *INFORMS Journal on Optimization* 1.2, pp. 164–183.

Bertsimas, Dimitris, Jack Dunn, and Aris Paschalidis (2017). "Regression and Classification using Optimal Decision Trees." In *2017 IEEE MIT undergraduate research technology conference*, pp. 1–4.

Bertsimas, Dimitris, Jack Dunn, and Yuchen Wang (2021). Near-optimal Nonlinear Regression Trees. *Operations Research Letters* 49.2, pp. 201–206.

Bessiere, Christian, Emmanuel Hebrard, and Barry O'Sullivan (2009). "Minimising decision tree size as combinatorial optimisation." In *International Conference on Principles and Practice of Constraint Programming*.

Bhatt, Rajen and Abhinav Dhall (2012). *Skin Segmentation Dataset*. UCI Machine Learning Repository.

Biggs, Max, Wei Sun, and Markus Ettl (2021). "Model distillation for revenue optimization: Interpretable personalized pricing." In *Proceedings of MLR-21*, pp. 946–956.

Binns, Reuben (2020). "On the apparent conflict between individual and group fairness." In *Proceedings of FAT* '20*.

Blanc, Guy, Jane Lange, Chirag Pabbaraju, Colin Sullivan, Li-Yang Tan, and Mo Tiwari (2023). "Harnessing the Power of Choices in Decision Tree Learning." In *Advances in NeurIPS-23*.

Blanquero, Rafael, Emilio Carrizosa, Cristina Molero-Río, and Dolores Romero Morales (2020). Sparsity in optimal randomized classification trees. *European Journal of Operational Research* 284.1, pp. 255–272.

Blanquero, Rafael, Emilio Carrizosa, Cristina Molero-Río, and Dolores Romero Morales (2021). Optimal randomized classification trees. *Computers & Operations Research* 132.

Blanquero, Rafael, Emilio Carrizosa, Cristina Molero-Río, and Dolores Romero Morales (2022). On Sparse Optimal Regression Trees. *European Journal of Operational Research* 299.3, pp. 1045–1054.

Blockeel, Hendrik, Laurens Devos, Benoît Frénay, Géraldin Nanfack, and Siegfried Nijssen (2023). Decision trees: from efficient prediction to responsible AI. *Frontiers in Artificial Intelligence* 6, p. 1124553.

Bock, R. (2007). *MAGIC Gamma Telescope Dataset*. UCI Machine Learning Repository.

Bos, Mim van den, Jacobus G. M. van der Linden, and Emir Demirović (2024). "Piecewise Constant and Linear Regression Trees: An Optimal Dynamic Programming Approach." In *Proceedings of ICML-24*, pp. 48994–49007.

Boutilier, Justin, Carla Michini, and Zachary Zhou (2023). Optimal multivariate decision trees. *Constraints* 28.4, pp. 549–577.

Breiman, Leo (2001). Statistical Modeling: The Two Cultures. *Statistical Science* 16.3, pp. 199–231.

Breiman, Leo, Jerome H. Friedman, Richard A. Olshen, and Charles J. Stone (1984). *Classification and Regression Trees*. Monterey, CA: Wadsworth and Brooks.

Brier, Glenn W. (1950). Verification of Forecasts Expressed in Terms of Probability. *Monthly Weather Review* 78.1, pp. 1–3.

Brița, Cătălin E., Jacobus G. M. van der Linden, and Emir Demirović (2025). "Optimal Classification Trees for Continuous Feature Data Using Dynamic Programming with Branch-and-Bound." In *Proceedings of AAAI-25*, pp. 11131–11139.

Brooks, Thomas, D. Pope, and Michael Marcolini (2014). *Airfoil Self-Noise*. UCI Machine Learning Repository.

Buntine, Wray and Tim Niblett (1992). A Further Comparison of Splitting Rules for Decision-Tree Induction. *Machine Learning* 8, pp. 75–85.

Buscema, M., S. Terzi, and W. Tastle (2010). *Steel Plates Faults Dataset*. UCI Machine Learning Repository.

Cabodi, Gianpiero, Paolo E. Camurati, Joao Marques-Silva, Marco Palena, and Paolo Pasini (2024). Optimizing Binary Decision Diagrams for Interpretable Machine Learning classification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 43.10, pp. 3083–3087.

Candanedo, Luis M. Ibarra and Veronique Feldheim (2016). Accurate occupancy detection of an office room from light, temperature, humidity and CO2 measurements using statistical learning models. *Energy and Buildings* 112, pp. 28–39.

Carreira-Perpinán, Miguel A. and Pooya Tavallali (2018). "Alternating Optimization of Decision Trees, with Application to Learning Sparse Oblique Trees." In *Advances in NeurIPS-18*.

Carrizosa, Emilio, Cristina Molero-Río, and Dolores Romero Morales (2021). Mathematical optimization in classification and regression trees. *TOP: An Official Journal of the Spanish Society of Statistics and Operations Research* 29.1, pp. 5–33.

Castro-Santos, Theodore and Alex Haro (2003). Quantifying migratory delay: a new application of survival analysis methods. *Canadian Journal of Fisheries and Aquatic Sciences* 60.8, pp. 986–996.

Caton, Simon and Christian Haas (2023). Fairness in Machine Learning: A Survey. *ACM Computing Surveys*.

Chaouki, Ayman, Jesse Read, and Albert Bifet (2025). "Branches: Efficiently Seeking Optimal Sparse Decision Trees via AO*." In *Proceedings of ICML-25*.

Chi, Chih-Lin, W. Nick Street, and William H. Wolberg (2007). "Application of Artificial Neural Network-Based Survival Analysis on Two Breast Cancer Datasets." In *Proceedings of the annual AMIA Symposium*, pp. 130–134.

Chipman, Hugh A., Edward I. George, and Robert E. McCulloch (1998). Bayesian CART Model Search. *Journal of the American Statistical Association* 93.443, pp. 935–948.

Chung, Ching-Fan, Peter Schmidt, and Ana D. Witte (1991). Survival Analysis: A Survey. *Journal of Quantitative Criminology* 7, pp. 59–98.

Ciampi, Antonio, Johanne Thiffault, Jean-Pierre Nakache, and Bernard Asselain (1986). Stratification by stepwise regression, correspondence analysis and recursive partition: a comparison of three methods of analysis for survival data with covariates. *Computational Statistics & Data Analysis* 4.3, pp. 185–204.

Ciaperoni, Martino, Han Xiao, and Aristides Gionis (2024). "Efficient Exploration of the Rashomon Set of Rule-Set Models." In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pp. 478–489.

Çınar, İlkay and Murat Koklu (2019). Classification of rice varieties using artificial intelligence methods. *International Journal of Intelligent Systems and Applications in Engineering* 7.3, pp. 188–194.

Çınar, İlkay, Murat Koklu, and Şakir Taşdemir (2020). Classification of raisin grains using machine vision and artificial intelligence methods. *Gazi Journal of Engineering Sciences* 6.3, pp. 200–209.

Cortez, Paulo and Alice Maria Gonçalves Silva (2008). "Using data mining to predict secondary school student performance." In *Proceedings of 5th FUture BUsiness TEChnology Conference*. Ed. by A. Brito and J. Teixeira, pp. 5–12.

Costa, Vinícius G. and Carlos E. Pedreira (2023). Recent Advances in Decision Trees: An Updated Survey. *Artificial Intelligence Review* 56, pp. 4765–4800.

Cox, David R. (1972). Regression Models and Life-Tables. *Journal of the Royal Statistical Society: Series B (Methodological)* 34.2, pp. 187–202.

Cox, Louis Anthony, Yuping Qiu, and Warren Kuehner (1989). Heuristic Least-Cost Computation of Discrete Classification Functions with Uncertain Argument Values. *Annals of Operations research* 21.1, pp. 1–29.

D'Amour, Alexander, Katherine Heller, Dan Moldovan, Ben Adlam, Babak Alipanahi, Alex Beutel, Christina Chen, Jonathan Deaton, Jacob Eisenstein, Matthew D. Hoffman, et al. (2022). Underspecification Presents Challenges for Credibility in Modern Machine Learning. *Journal of Machine Learning Research* 23.226, pp. 1–61.

D'Onofrio, Federico, Giorgio Grani, Marta Monaci, and Laura Palagi (2024). Margin optimal classification trees. *Computers & Operations Research* 161, p. 106441.

Davis, Roger B. and James R. Anderson (1989). Exponential survival trees. *Statistics in Medicine* 8.8, pp. 947–961.

De'ath, Glenn and Katharina E. Fabricius (2000). Classification and Regression Trees: A Powerful yet Simple Technique for Ecological Data Analysis. *Ecology* 81.11, pp. 3178–3192.

Demirović, Emir, Emmanuel Hebrard, and Louis Jean (2023). "Blossom: an Anytime Algorithm for Computing Optimal Decision Trees." In *Proceedings of ICML-23*, pp. 7533–7562.

Demirović, Emir, Anna Lukina, Emmanuel Hebrard, Jeffrey Chan, James Bailey, Christopher Leckie, Kotagiri Ramamohanarao, and Peter J. Stuckey (2022). MurTree: Optimal Decision Trees via Dynamic Programming and Search. *Journal of Machine Learning Research* 23.26, pp. 1–47.

Demirović, Emir and Peter J. Stuckey (2021). "Optimal Decision Trees for Nonlinear Metrics." In *Proceedings of AAAI-21*, pp. 3733–3741.

Demšar, Janez (2006). Statistical Comparisons of Classifiers over Multiple Data Sets. *Journal of Machine Learning Research* 7, pp. 1–30.

Denis, Christophe, Romuald Elie, Mohamed Hebiri, and François Hu (2021). Fairness guarantee in multi-class classification. *arXiv preprint arXiv:2109.13642*.

Denison, David G. T., Bani K. Mallick, and Adrian F. M. Smith (1998). A Bayesian CART Algorithm. *Biometrika* 85.2, pp. 363–377.

Detassis, Fabrizio, Michele Lombardi, and Michela Milano (2021). "Teaching the Old Dog New Tricks: Supervised Learning with Constraints." In *Proceedings of AAAI-21*.

Dietterich, Thomas G. (1995). Overfitting and Undercomputing in Machine Learning. *ACM Computing Surveys* 27.3, pp. 326–327.

Domingos, Pedro (1999). "Metacost: A General Method for Making Classifiers Cost-Sensitive." In *Proceedings of KDD-99*, pp. 155–164.

Donnelly, Jon, Srikar Katta, Cynthia Rudin, and Edward Browne (2023). "The Rashomon Importance Distribution: Getting RID of Unstable, Single Model-based Variable Importance." In *Advances in NeurIPS*. Vol. 36, pp. 6267–6279.

Drysdale, Erik (2022). SurvSet: An open-source time-to-event dataset repository. *arXiv preprint arXiv:2203.03094*.

Dua, Dheeru and Casey Graff (2017). *UCI Machine Learning Repository*. URL: `http://archive.ics.uci.edu/ml`.

Dudík, Miroslav, John Langford, and Lihong Li (2011). "Doubly Robust Policy Evaluation and Learning." In *Proceedings of ICML-11*, pp. 1097–1104.

Dunn, Jack William (2018). "Optimal Trees for Prediction and Prescription". PhD thesis. Massachusetts Institute of Technology.

Dunn, Olive Jean and Virginia A. Clark (2009). *Basic Statistics: A Primer for the Biomedical Sciences*. Hoboken, NJ: John Wiley & Sons.

Dutch Central Bureau for Statistics (2001a). *Dutch Census 2001 public use files (anonymized 1% samples from the microdata files)*. DANS. DOI: `https://doi.org/10.17026/dans-xms-xc4a`. URL: `https://easy.dans.knaw.nl/ui/datasets/id/easy-dataset:32357`.

Dutch Central Bureau for Statistics (2001b). *Dutch Census 2001 public use files (anonymized 1% samples from the microdata files)*. DOI: `https://doi.org/10.17026/dans-xms-xc4a`.

Dwork, Cynthia, Moritz Hardt, Toniann Pitassi, Omer Reingold, and Richard Zemel (2012). "Fairness Through Awareness." In *Proceedings of ICTS '12*, pp. 214–226.

Elmaghraby, Salah E. (1970). The Concept of "State" in Discrete Dynamic Programming. *Journal of Mathematical Analysis and Applications* 29.3, pp. 523–557.

Eryurt, Mehmet Ali and İsmet Koç (2012). Internal migration and fertility in Turkey: Kaplan-Meier survival analysis. *International Journal of Population Research* 2012.

Esposito, Floriana, Donato Malerba, and Giovanni Semeraro (1997). A Comparative Analysis of Methods for Pruning Decision Trees. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 19.5, pp. 476–491.

European Commission (2021). *Laying Down Harmonised Rules on Artificial Intelligence (Artificial Intelligence Act) and Amending Certain Union Legislative Acts COM(2021) 206*.

Feurer, Matthias, Jan N. Van Rijn, Arlind Kadra, Pieter Gijsbers, Neeratyoy Mallik, Sahithya Ravi, Andreas Müller, Joaquin Vanschoren, and Frank Hutter (2021). OpenML-Python: an extensible Python API for OpenML. *Journal of Machine Learning Research* 22.100, pp. 1–5.

FICO, Google, Imperial College London, MIT, University of Oxford, UC Irvine, and UC Berkeley (2018). *Explainable Machine Learning Challenge*. URL: `https://community.fico.com/s/explainable-machine-learning-challenge`.

Fisher, Aaron, Cynthia Rudin, and Francesca Dominici (2019). All Models are Wrong, but Many are Useful: Learning a Variable's Importance by Studying an Entire Class of Prediction Models Simultaneously. *Journal of Machine Learning Research* 20.177, pp. 1–81.

Flach, Peter (2012). *Machine Learning: The Art and Science of Algorithms that Make Sense of Data*. Cambridge: Cambridge University Press.

Fleming, Philip J. and John J. Wallace (1986). How not to lie with statistics: the correct way to summarize benchmark results. *Communications of the ACM* 29.3, pp. 218–221.

Florio, Alexandre M., Pedro Martins, Maximilian Schiffer, Thiago Serra, and Thibaut Vidal (2023). "Optimal Decision Diagrams for Classification." In *Proceedings AAAI-23*, pp. 7577–7585.

Frederickson, Greg N. and Donald B. Johnson (1982). The Complexity of Selection and Ranking in $X+Y$ and Matrices with Sorted Columns. *Journal of Computer and System Sciences* 24.2, pp. 197–208.

Freitas, Alberto, Altamiro Costa-Pereira, and Pavel Brazdil (2007). "Cost-Sensitive Decision Trees Applied to Medical Data." In *Proceedings of the International Conference on Data Warehousing and Knowledge Discovery (DaWaK-2007)*, pp. 303–312.

Freitas, Alex A. (2014). Comprehensible Classification Models – a position paper. *ACM SIGKDD Explorations Newsletter* 15.1, pp. 1–10.

Friedman, Jerome, Trevor Hastie, and Rob Tibshirani (2010). Regularization Paths for Generalized Linear Models via Coordinate Descent. *Journal of Statistical Software* 33.1, pp. 1–22.

Friedman, Jerome H. (1991). Multivariate adaptive regression splines. *The Annals of Statistics* 19.1, pp. 1–67.

Garey, Michael R. and Ronald L. Graham (1974). Performance Bounds on the Splitting Algorithm for Binary Testing. *Acta Informatica* 3.4, pp. 347–355.

Gerritsma, J., R. Onnink, and A. Versluis (2013). *Yacht Hydrodynamics*. UCI Machine Learning Repository.

Goodman, Rodney M. and Padhraic Smyth (1988). Decision Tree Design from a Communication Theory Standpoint. *IEEE Transactions on Information Theory* 34.5, pp. 979–994.

Gordon, Louis and Richard A. Olshen (1985). Tree-Structured Survival Analysis. *Cancer Treatment Reports* 69.10, pp. 1065–1069.

Graf, Erika, Claudia Schmoor, Willi Sauerbrei, and Martin Schumacher (1999). Assessment and comparison of prognostic classification schemes for survival data. *Statistics in Medicine* 18.17-18, pp. 2529–2545.

Grari, Vincent, Boris Ruf, Sylvain Lamprier, and Marcin Detyniecki (2020). Achieving fairness with decision trees: An adversarial approach. *Data Science and Engineering* 5.2, pp. 99–110.

Grinsztajn, Léo, Edouard Oyallon, and Gaël Varoquaux (2022). "Why do tree-based models still outperform deep learning on typical tabular data?" In *Advances in NeurIPS-22*.

Grubinger, Thomas, Achim Zeileis, and Karl-Peter Pfeiffer (2014). evtree: Evolutionary Learning of Globally Optimal Classification and Regression Trees in R. *Journal of Statistical Software* 61.1, pp. 1–29.

Günlük, Oktay, Jayant Kalagnanam, Minhan Li, Matt Menickelly, and Katya Scheinberg (2021). Optimal Decision Trees for Categorical Data via Integer Programming. *Journal of Global Optimization* 81, pp. 233–260.

Gurobi Optimization LLC (2022). *Gurobi Optimizer Reference Manual*. URL: `https://www.gurobi.com`.

Hara, Satoshi and Masakazu Ishihata (2018). "Approximate and Exact Enumeration of Rule Models." In *Proceedings of AAAI-18*, pp. 3157–3164.

Harrell, Frank E., Robert M. Califf, David B. Pryor, Kerry L. Lee, and Robert A. Rosati (1982). Evaluating the Yield of Medical Tests. *Jama* 247.18, pp. 2543–2546.

Harvey, William D. and Matthew L. Ginsberg (1995). "Limited Discrepancy Search." In *Proceedings of IJCAI-95*, pp. 607–615.

Hebrail, Georges and Alice Berard (2012). *Individual household electric power consumption*. UCI Machine Learning Repository.

Hemmateenejad, Bahram, Mojtaba Shamsipur, Vali Zare-Shahabadi, and Morteza Akhond (2011). Building optimal regression tree by ant colony system–genetic algorithm: Application to modeling of melting points. *Analytica Chimica Acta* 704, pp. 57–62.

Hothorn, Torsten, Kurt Hornik, and Achim Zeileis (2006). Unbiased Recursive Partitioning: A Conditional Inference Framework. *Journal of Computational and Graphical Statistics* 15.3, pp. 651–674.

Hu, Hao, Marie-José Huguet, and Mohamed Siala (2022). "Optimizing Binary Decision Diagrams with MaxSAT for Classification." In *Proceedings of AAAI-22*, pp. 3767–3775.

Hu, Hao, Mohamed Siala, Emmanuel Hebrard, and Marie-José Huguet (2020). "Learning Optimal Decision Trees with MaxSAT and its Integration in AdaBoost." In *IJCAI-PRICAI 2020*, pp. 1170–1176.

Hu, Qinghua, Xunjian Che, Lei Zhang, David Zhang, Maozu Guo, and Daren Yu (2011). Rank Entropy-Based Decision Trees for Monotonic Classification. *IEEE Transactions on Knowledge and Data Engineering* 24.11, pp. 2052–2064.

Hu, Xiyang, Cynthia Rudin, and Margo Seltzer (2019). "Optimal Sparse Decision Trees." In *Advances in NeurIPS-19*, pp. 7267–7275.

Hua, Kaixun, Jiayang Ren, and Yankai Cao (2022). "A Scalable Deterministic Global Optimization Algorithm for Training Optimal Decision Tree." In *Advances in NeurIPS-22*, pp. 8347–8359.

Huisman, Tim, Jacobus G. M. van der Linden, and Emir Demirović (2024). "Optimal Survival Trees: A Dynamic Programming Approach." In *Proceedings of AAAI-24*. Vol. 38. 11, pp. 12680–12688.

Hyafil, Laurent and Ronald L. Rivest (1976). Constructing optimal binary decision trees is NP-complete. *Information processing letters* 5.1, pp. 15–17.

Ignatiev, Alexey, Joao Marques-Silva, Nina Narodytska, and Peter J. Stuckey (2021). "Reasoning-based learning of interpretable ML models." In *Proceedings of IJCAI-21*, pp. 4458–4465.

Image Segmentation Dataset (1990). UCI Machine Learning Repository.

International Warfarin Pharmacogenetics Consortium (2009). Estimation of the Warfarin Dose with Clinical and Pharmacogenetic Data. *New England Journal of Medicine* 360.8, pp. 753–764.

Interpretable AI (2023). *Interpretable AI Documentation*. URL: `https://www.interpretable.ai/`.

Ishwaran, Hemant, Udaya B. Kogalur, Eugene H. Blackstone, and Michael S. Lauer (2008). Random Survival Forests. *The Annals of Applied Statistics* 2.3, pp. 841–860.

Izza, Yacine, Alexey Ignatiev, and Joao Marques-Silva (2022). On Tackling Explanation Redundancy in Decision Trees. *Journal of Artificial Intelligence Research* 75, pp. 261–321.

Janosi, Andras, William Steinbrunn, Matthias Pfisterer, and Robert Detrano (1988). *Heart Cleveland Dataset*. DOI: `https://doi.org/10.24432/C52P4X`. URL: `https://archive.ics.uci.edu/ml/datasets/heart+disease`.

Janota, Mikoláš and António Morgado (2020). "SAT-Based Encodings for Optimal Decision Trees with Explicit Paths." In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing (SAT 2020)*, pp. 501–518.

Jo, Nathanael, Sina Aghaei, Jack Benson, Andrés Gómez, and Phebe Vayanos (2023). "Learning Optimal Fair Decision Trees: Trade-offs Between Interpretability, Fairness, and Accuracy." In *Proceedings of the 2023 AAAI/ACM Conference on AI, Ethics, and Society*, pp. 181–192.

Jo, Nathanael, Sina Aghaei, Andrés Gómez, and Phebe Vayanos (2021). Learning Optimal Prescriptive Trees from Observational Data. *arXiv preprint arXiv:2108.13628*.

Johnson, Brian (2014). *Wilt Dataset*. UCI Machine Learning Repository.

Johnson, Donald B. and Tetsuo Mizoguchi (1978). Selecting the Kth Element in $X+Y$ and $X_1+X_2+\cdots+X_m$. *SIAM Journal on Computing* 7.2, pp. 147–153.

Justin, Nathan, Sina Aghaei, Andrés Gómez, and Phebe Vayanos (2022). "Optimal Robust Classification Trees." In *The AAAI-22 Workshop on Adversarial Machine Learning and Beyond*.

Kallus, Nathan (2017). "Recursive Partitioning for Personalization using Observational Data." In *Proceedings of ICML-17*, pp. 1789–1798.

Kamiran, Faisal and Toon Calders (2009). "Classifying without discriminating." In *2nd International Conference on Computer, Control and Communication*, pp. 333–338.

Kamiran, Faisal, Toon Calders, and Mykola Pechenizkiy (2010). "Discrimination Aware Decision Tree Learning." In *Proceedings of ICDM-10*, pp. 869–874.

Kaplan, Edward L. and Paul Meier (1958). Nonparametric Estimation from Incomplete Observations. *Journal of the American Statistical Association* 53.282, pp. 457–481.

Karmarkar, Narendra (1984). "A New Polynomial-Time Algorithm for Linear Programming." In *Proceedings of the sixteenth annual ACM symposium on Theory of Computing*, pp. 302–311.

Karp, Richard M. and Michael Held (1967). Finite-State Processes and Dynamic Programming. *SIAM Journal on Applied Mathematics* 15.3, pp. 693–718.

Kass, Gordon V. (1980). An Exploratory Technique for Investigating Large Quantities of Categorical Data. *Journal of the Royal Statistical Society. Series C (Applied Statistics)* 29.2, pp. 119–127.

Kearns, Michael and Yishay Mansour (1996). "On the Boosting Ability of Top-Down Decision Tree Learning Algorithms." In *Proceedings of the 28th Annual ACM symposium on Theory of Computing*, pp. 459–468.

Kendall, Maurice G. (1938). A New Measure of Rank Correlation. *Biometrika* 30.1-2, pp. 81–93.

King, Matthew W. and Patricia A. Resick (2014). Data Mining in Psychological Treatment Research: A Primer on Classification and Regression Trees. *Journal of Consulting and Clinical Psychology* 82.5, pp. 895–905.

Kingma, Diederik P. and Jimmy Ba (2015). "Adam: A Method for Stochastic Optimization." In *International Conference on Learning Representations*.

Kiossou, Harold, Pierre Schaus, and Siegfried Nijssen (2026). Anytime Optimal Decision Tree Learning with Continuous Features. *arXiv preprint arXiv:2601.14765*.

Kiossou, Harold, Pierre Schaus, Siegfried Nijssen, and Vinasetan Ratheil Houndji (2022). "Time constrained DL8.5 using Limited Discrepancy Search." In *Proceedings of ECML-PKDD-22*, pp. 443–459.

Kiossou, Harold Silvère, Siegfried Nijssen, and Pierre Schaus (2025). A Generic Complete Anytime Beam Search for Optimal Decision Tree. *arXiv preprint arXiv:2508.06064*.

Kobylińska, Katarzyna, Mateusz Krzyziński, Rafał Machowicz, Mariusz Adamek, and Przemysław Biecek (2024). Exploration of the Rashomon Set Assists

Trustworthy Explanations for Medical Data. *IEEE Journal of Biomedical and Health Informatics* 28.11, pp. 6454–6465.

Koklu, Murat and Ilker Ali Ozkan (2020). Multiclass classification of dry beans using computer vision and machine learning techniques. *Computers and Electronics in Agriculture* 174, p. 105507.

Kononenko, Igor (2001). Machine learning for medical diagnosis: history, state of the art and perspective. *Artificial Intelligence in Medicine* 23.1, pp. 89–109.

Kozodoi, Nikita, Johannes Jacob, and Stefan Lessmann (2022). Fairness in credit scoring: Assessment, implementation and profit implications. *European Journal of Operational Research* 297.3, pp. 1083–1094.

Krichevsky, Raphail E. and Victor K. Trofimov (1981). The Performance of Universal Encoding. *IEEE Transactions on Information Theory* 27.2, pp. 199–206.

Kruszewski, Paul (1996). "The Botanical Beauty of Random Binary Trees: A Method for the Synthetic Imagery of Botanical Trees". PhD thesis. Montreal: McGill University.

Kuzilek, Jakub, Martin Hlosta, and Zdenek Zdrahal (2017). Open university learning analytics dataset. *Scientific data* 4, p. 170171.

Laberge, Gabriel, Yann Pequignot, Alexandre Mathieu, Foutse Khomh, and Mario Marchand (2023). Partial Order in Chaos: Consensus on Feature Attributions in the Rashomon Set. *Journal of Machine Learning Research* 24.364, pp. 1–50.

Le Quy, Tai, Arjun Roy, Vasileios Iosifidis, Wenbin Zhang, and Eirini Ntoutsi (2022). A survey on datasets for fairness-aware machine learning. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, e1452.

LeBlanc, Michael and John Crowley (1992). Relative Risk Trees for Censored Survival Data. *Biometrics* 48.2, pp. 411–425.

LeBlanc, Michael and John Crowley (1993). Survival Trees by Goodness of Split. *Journal of the American Statistical Association* 88.422, pp. 457–467.

Lei, Jing, Max G'Sell, Alessandro Rinaldo, Ryan J Tibshirani, and Larry Wasserman (2018). Distribution-free predictive inference for regression. *Journal of the American Statistical Association* 113.523, pp. 1094–1111.

Lemaire, Valentin, Gaël Aglin, and Siegfried Nijssen (2024). "Interpretable Quantile Regression by Optimal Decision Trees." In *International Symposium on Intelligent Data Analysis*, pp. 210–222.

Li, Duan and Yacov Y. Haimes (1991). Extension of Dynamic Programming to Nonseparable Dynamic Optimization Problems. *Computers & Mathematics with Applications* 21.12-12, pp. 51–56.

Li, Ming and Paul Vitányi (2008). *An Introduction to Kolmogorov Complexity and Its Applications*. 3rd. New York, NY: Springer.

Lin, Jimmy, Chudi Zhong, Diane Hu, Cynthia Rudin, and Margo Seltzer (2020). "Generalized and Scalable Optimal Sparse Decision Trees." In *Proceedings of ICML-20*, pp. 6150–6160.

Linardatos, Pantelis, Vasilis Papastefanopoulos, and Sotiris Kotsiantis (2020). Explainable AI: A Review of Machine Learning Interpretability Methods. *Entropy* 23.1, p. 18.

Linden, Jacobus G. M. van der, Daniël Vos, Mathijs M. de Weerdt, Sicco Verwer, and Emir Demirović (2024). Optimal or Greedy Decision Trees? Revisiting their Objectives, Tuning, and Performance. *arXiv preprint arXiv:2409.12788*.

Linden, Jacobus G. M. van der, Mathijs M. de Weerdt, and Emir Demirović (2022). "Fair and Optimal Decision Trees: A Dynamic Programming Approach." In *Advances in NeurIPS-22*.

Linden, Jacobus G. M. van der, Mathijs M. de Weerdt, and Emir Demirović (2023). "Necessary and Sufficient Conditions for Optimal Decision Trees using Dynamic Programming." In *Advances in NeurIPS-23*, pp. 9173–9212.

Ling, Charles X., Victor S. Sheng, and Qiang Yang (2006). Test Strategies for Cost-Sensitive Decision Trees. *IEEE Transactions on Knowledge and Data Engineering* 18.8, pp. 1055–1067.

Liu, Enhao, Tengmu Hu, Theodore T. Allen, and Christoph Hermes (2024). Optimal classification trees with leaf-branch and binary constraints. *Computers & Operations Research* 166, p. 106629.

Liu, Jiachang, Chudi Zhong, Boxuan Li, Margo Seltzer, and Cynthia Rudin (2022). "FasterRisk: Fast and Accurate Interpretable Risk Scores." In *Advances in NeurIPS-22*, pp. 17760–17773.

Loh, Wei-Yin (2002). Regression Trees with Unbiased Variable Selection and Interaction Detection. *Statistica Sinica* 12.2, pp. 361–386.

Loh, Wei-Yin (2014). Fifty Years of Classification and Regression Trees. *International Statistical Review* 82.3, pp. 329–348.

Lohweg, Volker (2013). *Banknote Authentication Dataset*. UCI Machine Learning Repository.

Lomax, Susan and Sunil Vadera (2013). A Survey of Cost-Sensitive Decision Tree Induction Algorithms. *ACM Computing Surveys* 45.2, pp. 1–35.

Lomax, Susan and Sunil Vadera (2017). A Cost-Sensitive Decision Tree Learning Algorithm based on a Multi-Armed Bandit Framework. *The Computer Journal* 60.7, pp. 941–956.

Lomax, Susan Elaine (2013). "Cost-Sensitive Decision Tree Learning using a Multi-Armed Bandit Framework". PhD thesis. University of Salford.

Lyon, R. J., B. W. Stappers, S. Cooper, J. M. Brooke, and J. D. Knowles (2016). Fifty years of pulsar candidate selection: from simple filters to a new principled real-time classification approach. *Monthly Notices of the Royal Astronomical Society* 459.1, pp. 1104–1123.

Malerba, Donato (1995). *Page Blocks Classification Dataset*. UCI Machine Learning Repository.

Maliah, Shlomi and Guy Shani (2021). Using POMDPs for learning cost sensitive decision trees. *Artificial Intelligence* 292, p. 103400.

Mangasarian, O. L. and W. H. Wolberg (1990). Cancer Diagnosis via Linear Programming. *SIAM News* 23.5, pp. 1–18.

Martelli, Alberto and Ugo Montanari (1978). Optimizing Decision Trees Through Heuristically Guided Search. *Communications of the ACM* 21.12, pp. 1025–1039.

Marton, Sascha, Stefan Lüdtke, Christian Bartelt, and Heiner Stuckenschmidt (2024). "GradTree: Learning Axis-Aligned Decision Trees with Gradient Descent." In *Proceedings of AAAI-24*.

Marx, Charles T., Flavio du Pin Calmon, and Berk Ustun (2020). "Predictive Multiplicity in Classification." In *Proceedings of ICML-20*, pp. 6765–6774.

Mazumder, Rahul, Xiang Meng, and Haoyue Wang (2022). "Quant-BnB: A Scalable Branch-and-Bound Method for Optimal Decision Trees with Continuous Features." In *Proceedings of ICML-22*, pp. 15255–15277.

McTavish, Hayden, Chudi Zhong, Reto Achermann, Ilias Karimalis, Jacques Chen, Cynthia Rudin, and Margo Seltzer (2022). "Fast Sparse Decision Tree Optimization via Reference Ensembles." In *Proceedings of AAAI-22*, pp. 9604–9613.

Mehrabi, Ninareh, Fred Morstatter, Nripsuta Saxena, Kristina Lerman, and Aram Galstyan (2021). A Survey on Bias and Fairness in Machine Learning. *ACM Computing Surveys* 54.6, pp. 1–35.

Mehta, Manish, Jorma Rissanen, and Rakesh Agrawal (1995). "MDL-based Decision Tree Pruning." In *Proceedings of KDD-95*, pp. 216–221.

Min, Fan and William Zhu (2012). "A Competition Strategy to Cost-Sensitive Decision Trees." In *Proceedings of the International Conference on Rough Sets and Knowledge Technology (RSKT)*, pp. 359–368.

Mingers, John (1989a). An Empirical Comparison of Pruning Methods for Decision Tree Induction. *Machine Learning* 4, pp. 227–243.

Mingers, John (1989b). An Empirical Comparison of Selection Measures for Decision-Tree Induction. *Machine Learning* 3, pp. 319–342.

Miyakawa, Masahiro (1985). Optimum Decision Trees - An Optimal Variable Theorem and its Related Applications. *Acta Informatica* 22, pp. 475–498.

Molinaro, Annette M., Sandrine Dudoit, and Mark J. Van der Laan (2004). Tree-based multivariate regression and density estimation with right-censored data. *Journal of Multivariate Analysis* 90.1, pp. 154–177.

Morgan, James N. and John A. Sonquist (1963). Problems in the Analysis of Survey Data, and a Proposal. *Journal of the American Statistical Association* 58.302, pp. 415–434.

Moro, Sérgio, Paulo Cortez, and Paulo Rita (2014). A data-driven approach to predict the success of bank telemarketing. *Decision Support Systems* 62, pp. 22–31.

Murphy, Owen J. and R. L. McCraw (1991). Designing Storage Efficient Decision Trees. *IEEE Transactions on Computers* 40.3, pp. 315–320.

Murthy, Sreerama K. and Steven Salzberg (1995). "Decision Tree Induction: How Effective Is the Greedy Heuristic?" In *Proceedings of KDD-95*, pp. 222–227.

Nanfack, Géraldin, Paul Temple, and Benoît Frénay (2022). Constraint Enforcement on Decision Trees: A Survey. *ACM Computing Surveys* 54.10, pp. 1–36.

Narodytska, Nina, Alexey Ignatiev, Filipe Pereira, and João Marques-Silva (2018). "Learning Optimal Decision Trees with SAT." In *Proceedings of IJCAI-18*, pp. 1362–1368.

Nelson, Wayne (1972). Theory and Applications of Hazard Plotting for Censored Failure Data. *Technometrics* 14.4, pp. 945–966.

Niblett, Tim (1987). "Constructing Decision Trees in Noisy Domains." In *Proceedings of the 2nd European Conference on European Working Session on Learning*, pp. 67–78.

Nijssen, Siegfried and Elisa Fromont (2007). "Mining Optimal Decision Trees from Itemset Lattices." In *Proceedings of SIGKDD-07*, pp. 530–539.

Nijssen, Siegfried and Elisa Fromont (2010). Optimal constraint-based decision tree induction from itemset lattices. *Data Mining and Knowledge Discovery* 21.1, pp. 9–51.

Noel, Mathew Mithra, Arindam Banerjee, Geraldine Bessie Amali D., and Venkataraman Muthiah-Nakarajan (2023). Alternate Loss Functions for Classification and Robust Regression Can Improve the Accuracy of Artificial Neural Networks. *arXiv:2303.09935*.

Norton, Steven W. (1989). "Generating Better Decision Trees." In *Proceedings of IJCAI-89*, pp. 800–805.

Núñez, Marlon (1991). The Use of Background Knowledge in Decision Tree Induction. *Machine Learning* 6.3, pp. 231–250.

Oates, Tim and David Jensen (1997). The Effects of Training Set Size on Decision Tree Complexity. *Sixth International Workshop on Artificial Intelligence and Statistics*, pp. 379–390.

Ordoni, Elaheh, Jakob Bach, and Ann-Katrin Fleck (2022). *Auction Verification*. UCI Machine Learning Repository.

Ordyniak, Sebastian and Stefan Szeider (2021). "Parameterized Complexity of Small Decision Tree Learning." In *Proceedings of AAAI-21*, pp. 6454–6462.

Patil, Dipti D., V. M. Wadhai, and J. A. Gokhale (2010). Evaluation of Decision Tree Pruning Algorithms for Complexity and Classification Accuracy. *International Journal of Computer Applications* 11.2, pp. 23–30.

Payne, Harold J. and William S. Meisel (1977). An Algorithm for Constructing Optimal Binary Decision Trees. *IEEE Transactions on Computers* C-26.9, pp. 905–916.

Pedreschi, Dino, Salvatore Ruggieri, and Franco Turini (2008). "Discrimination-aware Data Mining." In *Proceedings of SIGKDD-08*, pp. 560–568.

Pei, Shenglei, Qinghua Hu, and Chao Chen (2016). Multivariate decision trees with monotonicity constraints. *Knowledge-Based Systems* 112, pp. 14–25.

Piccialli, Veronica, Dolores Romero Morales, and Cecilia Salvatore (2024). Supervised feature compression based on counterfactual analysis. *European Journal of Operational Research* 317.2, pp. 273–285.

Piltaver, Rok, Mitja Luštrek, Matjaž Gams, and Sandra Martinčić-Ipšić (2016). What makes classification trees comprehensible? *Expert Systems with Applications* 62, pp. 333–346.

Quinlan, J. Ross (1986). Induction of Decision Trees. *Machine Learning* 1.1, pp. 81–106.

Quinlan, J. Ross (1987). Simplifying Decision Trees. *International Journal of Man-Machine Studies* 27.3, pp. 221–234.

Quinlan, J. Ross (1992). "Learning with continuous classes." In *Proceedings of the 5th Australian Joint Conference on Artificial Intelligence*, pp. 343–348.

Quinlan, J. Ross (1993a). *Auto MPG*. UCI Machine Learning Repository.

Quinlan, J. Ross (1993b). *C4.5: Programs for Machine Learning*. San Francisco, CA: Morgan Kaufmann Publishers Inc.

Quinlan, J. Ross and R. M. Cameron-Jones (1995). "Oversearching and Layered Search in Empirical Learning." In *Proceedings of IJCAI-95*, pp. 1019–1024.

Quinlan, J. Ross and Ronald L. Rivest (1989). Inferring Decision Trees using the Minimum Description Length Principle. *Information and Computation* 80.3, pp. 227–248.

Raileanu, Laura Elena and Kilian Stoffel (2004). Theoretical comparison between the Gini Index and Information Gain criteria. *Annals of Mathematics and Artificial Intelligence* 41, pp. 77–93.

Ranzato, Francesco, Caterina Urban, and Marco Zanella (2021). Fair Training of Decision Tree Classifiers. *arXiv preprint arXiv:2101.00909*.

Rissanen, Jorma (1978). Modeling by Shortest Data Description. *Automatica* 14.5, pp. 465–471.

Rissanen, Jorma (1997). Stochastic Complexity in Learning. *Journal of Computer and System Sciences* 55.1, pp. 89–95.

Roesler, Oliver (2013). *EEG Eye State Dataset*. UCI Machine Learning Repository.

Rudin, Cynthia (2019). Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead. *Nature Machine Intelligence* 1.5, pp. 206–215.

Rudin, Cynthia, Chaofan Chen, Zhi Chen, Haiyang Huang, Lesia Semenova, and Chudi Zhong (2022). Interpretable machine learning: Fundamental principles and 10 grand challenges. *Statistic Surveys* 16, pp. 1–85.

Rudin, Cynthia, Chudi Zhong, Lesia Semenova, Margo Seltzer, Ronald Parr, Jiachang Liu, Srikar Katta, Jon Donnelly, Harry Chen, and Zachery Boner (2024). "Position: Amazing Things Come From Having Many Good Models." In *Proceedings of ICML-24*. Vol. 235. Proceedings of Machine Learning Research, pp. 42783–42795.

Schumacher, Helmut and Kenneth C. Sevcik (1976). The Synthetic Approach to Decision Table Conversion. *Communications of the ACM* 19.6, pp. 343–351.

Segal, Mark Robert (1988). Regression Trees for Censored Data. *Biometrics* 44.1, pp. 35–47.

Select Committee on Artificial Intelligence of the National Science & Technology Council (2019). *The National Artifical Intelligence Research and Development Strategic Plan: 2019 Update*. Tech. rep. URL: `https://www.nitrd.gov/pubs/National-AI-RD-Strategy-2019.pdf`.

Selvin, Steve (2008). *Survival Analysis for Epidemiologic and Medical Research*. Cambridge: Cambridge University Press.

Semenova, Lesia, Cynthia Rudin, and Ronald Parr (2022). "On the Existence of Simpler Machine Learning Models." In *Proceedings of the 2022 ACM Conference on Fairness, Accountability, and Transparency*, pp. 1827–1858.

Shati, Pouya, Eldan Cohen, and Sheila McIlraith (2021). "SAT-Based Approach for Learning Optimal Decision Trees with Non-Binary Features." In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP-2021)*, 50:1–50:16.

Shati, Pouya, Eldan Cohen, and Sheila McIlraith (2023a). "SAT-Based Learning of Compact Binary Decision Diagrams for Classification." In *Proceedings of the 29th International Conference on Principles and Practice of Constraint Programming (CP 2023)*, pp. 1–33.

Shati, Pouya, Eldan Cohen, and Sheila A. McIlraith (2023b). SAT-based optimal classification trees for non-binary data. *Constraints* 28.2, pp. 166–202.

Shati, Pouya, Yuliang Song, Eldan Cohen, and Sheila McIlraith (2025). Optimal Decision Trees for Interpretable and Constrained Clustering. *Journal of Artificial Intelligence Research* 84, pp. 1–4.

Shih, Yu-Shan (1999). Families of splitting criteria for classification trees. *Statistics and Computing* 9.4, pp. 309–315.

Shill Bidding Dataset (2020). UCI Machine Learning Repository.

Singh, Adarsh Pal, Vivek Jain, Sachin Chaudhari, Frank Alexander Kraemer, Stefan Werner, and Vishal Garg (2018). "Machine learning-based occupancy estimation using multivariate sensor nodes." In *2018 IEEE Globecom Workshops*, pp. 1–6.

Song, Mingzhou and Hua Zhong (2020). Efficient weighted univariate clustering maps outstanding dysregulated genomic zones in human cancers. *Bioinformatics* 36.20, pp. 5027–5036.

Staus, Luca Pascal, Christian Komusiewicz, Frank Sommer, and Manuel Sorge (2025). "Witty: An Efficient Solver for Computing Minimum-Size Decision Trees." In *Proceedings of AAAI-25*, pp. 20584–20591.

Stefano, Claudio, Francesco Fontanella, Marilena Maniaci, and Alessandra Freca (2018). *Avila Dataset*. UCI Machine Learning Repository.

Strack, Beata, Jonathan P. DeShazo, Chris Gennings, Juan L. Olmo, Sebastian Ventura, Krzysztof J. Cios, and John N. Clore (2014). Impact of HbA1c

measurement on hospital readmission rates: analysis of 70,000 clinical database patient records. *BioMed research international*.

Struyf, Jan and Sašo Džeroski (2007). "Clustering Trees with Instance Level Constraints." In *Proceedings of ECML-07*, pp. 359–370.

Su, Xiaogang and Juanjuan Fan (2004). Multivariate Survival Trees: A Maximum Likelihood Approach Based on Frailty Models. *Biometrics* 60.1, pp. 93–99.

Subramanian, Shivaram, Wei Sun, Youssef Drissi, and Markus Ettl (2022). "Constrained Prescriptive Trees via Column Generation." In *Proceedings of AAAI-22*, pp. 4602–4610.

Sullivan, Colin, Mo Tiwari, and Sebastian Thrun (2024). "MAPTree: Beating "Optimal" Decision Trees with Bayesian Decision Trees." In *Proceedings of AAAI-24*.

Tan, Ming (1993). Cost-Sensitive Learning of Classification Knowledge and its Applications in Robotics. *Machine Learning* 13.1, pp. 7–33.

Therneau, Terry, Beth Atkinson, and Brian Ripley (2023). *Package 'rpart'*.

Therneau, Terry M., Patricia M. Grambsch, and Thomas R. Fleming (1990). Martingale-based residuals for survival models. *Biometrika* 77.1, pp. 147–160.

Tsallis, Constantino (1988). Possible Generalization of Boltzmann-Gibbs Statistics. *Journal of Statistical Physics* 52, pp. 479–487.

Tsanas, Athanasios and Angeliki Xifara (2012). *Energy efficiency*. UCI Machine Learning Repository.

Turing Institute (Glasgow) (1987). *Statlog (Vehicle Silhouettes) Dataset*. URL: `https://archive.ics.uci.edu/ml/datasets/Statlog+%28Vehicle+Silhouettes%29`.

Turney, Peter D. (1995). Cost-Sensitive Classification: Empirical Evaluation of a Hybrid Genetic Decision Tree Induction Algorithm. *Journal of artificial intelligence research* 2, pp. 369–409.

Turney, Peter D. (2000). "Types of Cost in Inductive Concept Learning." In *Proceedings of the ICML'2000 Workshop on Cost-Sensitive Learning*, pp. 15–21.

Ulrich, Karl (1993). *Servo*. UCI Machine Learning Repository.

Valdivia, Ana, Javier Sánchez-Monedero, and Jorge Casillas (2021). How fair can we go in machine learning? Assessing the boundaries of accuracy and fairness. *International Journal of Intelligent Systems* 36.4, pp. 1619–1643.

Van Belle, Vanya, Kristiaan Pelckmans, Sabine Van Huffel, and Johan A. K. Suykens (2011). Support vector methods for survival analysis: a comparison between ranking and regression approaches. *Artificial Intelligence in Medicine* 53.2, pp. 107–118.

Vanschoren, Joaquin, Jan N. Van Rijn, Bernd Bischl, and Luis Torgo (2013). OpenML: networked science in machine learning. *SIGKDD Explorations* 15.2, pp. 49–60.

Verhaeghe, Hélene, Siegfried Nijssen, Gilles Pesant, Claude-Guy Quimper, and Pierre Schaus (2020). Learning Optimal Decision Trees using Constraint Programming. *Constraints* 25.3, pp. 226–250.

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

Verwer, Sicco and Yingqian Zhang (2019). "Learning Optimal Classification Trees Using a Binary Linear Program Formulation." In *Proceedings of AAAI-19*, pp. 1625–1632.

Vilas Boas, Matheus Guedes, Haroldo Gambini Santos, Luiz Henrique de Campos Merschmann, and Greet Vanden Berghe (2021). Optimal Decision Trees for the Algorithm Selection Problem: Integer Programming Based Approaches. *International Transactions in Operational Research* 28.5, pp. 2759–2781.

Vos, Daniël and Sicco Verwer (2022). "Robust Optimal Classification Trees against Adversarial Examples." In *Proceedings of AAAI-22*, pp. 8520–8528.

Vos, Daniël and Sicco Verwer (2023). "Optimal Decision Tree Policies for Markov Decision Processes." In *Proceedings of IJCAI-23*, pp. 5457–5465.

Vreeken, Jilles, Matthijs van Leeuwen, and Arno Siebes (2011). Krimp: mining itemsets that compress. *Data Mining and Knowledge Discovery* 23.1, pp. 169–214.

Wang, Haizhou and Mingzhou Song (2011). Ckmeans.1d.dp: optimal k-means clustering in one dimension by dynamic programming. *The R Journal* 3.2, pp. 29–33.

Wang, Jingbo, Yannan Li, and Chao Wang (2022). "Synthesizing Fair Decision Trees via Iterative Constraint Solving." In *Proceedings of the International Conference on Computer Aided Verification*, pp. 364–385.

Wang, Ping, Yan Li, and Chandan K. Reddy (2019). Machine Learning for Survival Analysis: A Survey. *ACM Computing Surveys (CSUR)* 51.6, pp. 1–36.

Wang, Tong, Cynthia Rudin, Finale Doshi-Velez, Yimin Liu, Erica Klampfl, and Perry MacNeille (2017). A bayesian framework for learning rule sets for interpretable classification. *Journal of Machine Learning Research* 18.70, pp. 1–37.

Wang, Yisen and Shu-Tao Xia (2017). "Unifying attribute splitting criteria of decision trees by Tsallis entropy." In *Proceedings of the International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 507–2511.

Wang, Yong and Ian H. Witten (1997). "Induction of Model Trees for Predicting Continuous Classes." In *Proceedings of the European Conference on Machine Learning*.

Xin, Rui, Chudi Zhong, Zhi Chen, Takuya Takagi, Margo Seltzer, and Cynthia Rudin (2022). "Exploring the Whole Rashomon Set of Sparse Decision Trees." In *Advances in NeurIPS-22*, pp. 14071–14084.

Yang, Lingjian, Songsong Liu, Sophia Tsoka, and Lazaros G. Papageorgiou (2016). Mathematical Programming for Piecewise Linear Regression Analysis. *Expert Systems with Applications* 44, pp. 156–167.

Yang, Lingjian, Songsong Liu, Sophia Tsoka, and Lazaros G. Papageorgiou (2017). A regression tree approach using mathematical programming. *Expert Systems with Applications* 78, pp. 347–357.

Zhang, Rui, Rui Xin, Margo Seltzer, and Cynthia Rudin (2023). "Optimal Sparse Regression Trees." In *Proceedings of AAAI-23*, pp. 11270–11279.

Zhang, Rui, Rui Xin, Margo Seltzer, and Cynthia Rudin (2024). "Optimal Sparse Survival Trees." In *Proceedings of The 27th International Conference on Artificial Intelligence and Statistics*, pp. 352–360.

Zhang, Weixiong (1998). Complete Anytime Beam Search. *Proceedings of AAAI-98*, pp. 425–430.

Zharmagambetov, Arman, Suryabhan Singh Hada, Magzhan Gabidolla, and Miguel A. Carreira-Perpiñán (2021). "Non-Greedy Algorithms for Decision Tree Optimization: An Experimental Comparison." In *2021 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8.

Zhong, Chudi, Zhi Chen, Jiachang Liu, Margo Seltzer, and Cynthia Rudin (2023). "Exploring and Interacting with the Set of Good Sparse Generalized Additive Models." In *Advances in NeurIPS-23*, pp. 56673–56699.

Zhou, Zhengyuan, Susan Athey, and Stefan Wager (2022). Offline Multi-Action Policy Learning: Generalization and Optimization. *Operations Research* 71.1, pp. 148–183.

Zhu, Chloe Qinyu, Muhang Tian, Lesia Semenova, Jiachang Liu, Jack Xu, Joseph Scarpa, and Cynthia Rudin (2025). Fast and Interpretable Mortality Risk Scores for Critical Care Patients. *Journal of the American Medical Informatics Association* 32.4, pp. 736–747.

Zhu, Haoran, Pavankumar Murali, Dzung T. Phan, Lam M. Nguyen, and Jayant R. Kalagnanam (2020). "A Scalable MIP-based Method for Learning Optimal Multivariate Decision Trees." In *Advances in NeurIPS-20*, pp. 1771–1781.

Zubek, Valentina Bayer and Thomas G. Dietterich (2002). "Pruning Improves Heuristic Search for Cost-Sensitive Learning." In *Proceedings of ICML-02*, pp. 19–26.

Zwitter, M. and M. Soklic (1988a). *Lymphography Dataset*. URL: `https://archive.ics.uci.edu/ml/datasets/Lymphography`.

Zwitter, M. and M. Soklic (1988b). *Primary Tumor Dataset*. URL: `https://archive.ics.uci.edu/ml/datasets/primary+tumor`.

Zwitter, M. and M. Soklic (n.d.). *Breast Cancer Dataset*. DOI: `https://doi.org/10.24432/C51P4M`. URL: `https://archive.ics.uci.edu/ml/datasets/Breast+Cancer`.

Acknowledgements

As a concluding personal statement, I want to express my gratitude over this PhD journey. Throughout the whole process, I have continually felt blessed and grateful for the people who supported me, for the joy of solving such interesting problems, for the beauty in mathematics, and for the new skills I learned, specifically in writing.

First, I want to express my gratitude to my supervisors Mathijs de Weerdt and Emir Demirović. **Mathijs**, you have been my supervisor for nine (almost ten) years, starting with supervising my master thesis project, then for the electric vehicle charging and train maintenance projects, and finally as promotor during my PhD journey. I am grateful for how you lead the algorithmics group and how you have been my supervisor, for your kindness, for clear feedback, for the opportunities you gave, for the flexibility. You have been a great supervisor to me over all those years, and you were one of the reasons why I chose to stay for such a long time with this group. **Emir**, I am also grateful for all your support and all your involvement in our projects, and for the independence that you gave me. Your enthusiasm for combinatorial optimization and exact solutions is inspiring, and it has been great to brainstorm together with you about solutions and develop new ideas. I specifically appreciate all the positive and extensive constructive feedback you have given on my writing, which has helped a lot in improving my writing skills. When you won the CP early career award at the CP conference last year, I was not surprised, because I know that you have a deep understanding of optimization, work hard, and aim for high quality in your work. I do not think I could have wished for a better supervisor, and I am grateful to you for that.

Second, I want to express my gratitude to my colleagues. **Konstantin**, thank you for all the great conversations we have had over the years, about communism, chess, math, faith, and many other topics. I admire your principled approach to life and science, and also of course your incredible understanding of math. I still think you do not always realize that not everyone is as smart as you are! **Daniël**, you have been my favorite fellow tree researcher, and I enjoyed all the conversations we could have about decision tree research. I also admire your deep insight into machine learning. But even more, I appreciate what you added to our research group since you joined. I am grateful for how your presence has positively changed the way we interact as a group. **Elif**, it was a pleasure to research together with you and our long debugging sessions are something to remember. I am also grateful for the deeper conversations we could have about faith, and I hope we can one day agree on this topic! **Lei**, thank you for your friendship and all the conversations! **Longjian**, you are the most polite and kind person that I know. **Eggie**, a pity that you have lost your royal status! Thank you for all the morning tea times we shared. **Grisha**, thank you for all the conversations that spiraled out into so many different directions every time. **Sterre**, I wonder if you are predetermined to read or not read

this text, but I freely choose to express my gratitude for all our conversations on work, life, and logic. **Qisong** and **Yang**, thank you for all the meals and hotpot that we could share. And thanks also to all my other colleagues over the years in the algorithm group. I am grateful that we could be a group with so much kindness and friendliness, openness, and willingness to help each other.

Third, I want to express my gratitude to the bachelor and master students I supervised. **Mim**, great to work together with you both for the bachelor and master thesis project. Brainstorming together with you and discovering new intuitions as to what does and does not work has been most enjoyable. I am also grateful for publishing together with you. **Catalin** and **Tim**, thank you as well for our collaboration, your insights, and all the work you have done. I am also grateful that we could publish our research together.

Fourth, I want to express my gratitude to the scientific community. We are truly "standing on the shoulders of giants" as Google scholar reminds us. It is great to see a global scientific community that together pursues the common good, shares ideas, values integrity, and advances knowledge (albeit with many shortcomings). Specifically, I want to thank **Siegfried Nijssen** and **Pierre Schaus** for inviting me both to UCLouvain and KU Leuven. Both visits were highlights of my PhD, and I still remember the long and fruitful brainstorm sessions that we had.

Fifth, I want to express my deep gratitude to my friends and church community. **Coos** and **Elisabeth**, thank you for all your support, for all your prayers, wisdom, and conversations. And thanks to **Hannah-Louise**: it is such a delight that a joyful baby knows my name and is happy to see me. **Mattijs**, thank you as well for all our walks, talks, trips to Ukraine, and prayer, and in sharing some of the same burden. **Paul**, your friendship has been so important to me, both in the time you lived here and after you went back. I am deeply grateful for all your support and encouragement. I also want to thank all those who joined the navigator group over the last couple of years. **Mini**, thank you for your friendship. You have brought so much joy to our group and your laughter is contagious. **Yiyun**, thank you for all your friendship and care. **Cehao**, thank you for your friendship, and everything you taught me about the beauty of light and colors. **Qi**, thank you for all our talks about books, life, and faith. You are great at asking questions! **Rhodé**, great to see how you always invest in new people. **Isaac**, your generosity is inspiring. Also, I am grateful for Sinian, Jeroen, Genjing, Juan, Lina, Steffi, Elias, Micah, Yiyi, Jae, Gareth, Franz, Benjamin, Deborah, Alexia, Espen, Zander, Tom, Joan, Javier, Yarai, Sanmi, Yu-Chen, Mohit, Nitisha, David, Cindy, Carlos, Sándor, Anamaria, Levi, Sander, Keyang, and Kangmin, and many others. I am grateful for my church, Redeemer, for **Dean** and **Floyd**, in supporting me, **Daniel** and **Marie-Eve**, for hosting me so many times, and **Rico**, **Kelly**, **Wessel**, and many others for all the great meeting nights together. And other friends as well: **Elbert**, I am grateful for our fourteen-year friendship and I admire your zeal. **Jorge**, **Alison**, and **Sophie**, thanks for all the time together. **Joshua** and **Leoné**, thank you for your friendship over these years. **Pieter**, good to have had you as my housemate for so long.

Sixth, I want to express my gratitude to my family. To my **father** and **mother**, who have raised me, taught me wisdom about life, about God, about sacrifice, and

the value of perseverance. Thanks to my mother for all the sacrifices she made, and for the support of my father and all his prayers. To my sisters, **Rianne**, **Marlies**, and **Erika**, thank you for your care!

Last, and most importantly, I am grateful to **God**, since "every good gift and every perfect gift is from above, coming down from the Father of lights" (James 1:17). Therefore, all the gratitude expressed above is ultimately also gratitude toward God. *Soli Deo Gloria*.

Koos

Curriculum Vitæ

Jacobus G. M. van der Linden

21-04-1993 Born in Klundert, the Netherlands.

Education

2011–2014 BSc in Computer Science (*cum laude*)
Delft University of Technology
Minor Applied Physics
Thesis internship at Infotron, Amsterdam

2014–2017 MSc in Computer Science
Delft University of Technology
Thesis "Decision diagrams for decomposed mixed integer linear programs"

2020–Now MA in Theology and Religious Studies (ongoing)
Evangelical Theological Faculty, Leuven, Belgium
Formative Sources Track

2021–2026 PhD in Computer Science
Delft University of Technology

Work Experience

2017–2021 Student Coach
Navigators, Driebergen

2017–2021 Scientific Programmer
Delft University of Technology

List of Publications

Publications and preprints related to the PhD study

[1] E. Arslan, **J.G.M. van der Linden**, S. Hoogendoorn, M. Rinaldi, and E. Demirović, "SORTeD Rashomon Sets of Sparse Decision Trees: Anytime Enumeration," in *Advances in NeurIPS* (2025).

[2] C.E. Brita, **J.G.M. van der Linden**, and E. Demirović, "Optimal Classification Trees for Continuous Feature Data Using Dynamic Programming with Branch-and-Bound," in *Proceedings of AAAI*, 11131-11139 (2025).

[3] **J.G.M. van der Linden**, D. Vos, M.M. de Weerdt, S. Verwer and E. Demirović, "Optimal or Greedy Decision Trees? Revisiting their Objectives, Tuning, and Performance," *arXiv preprint arXiv:2409.12788* (2024).

[4] M. van den Bos, **J.G.M. van der Linden**, and E. Demirović, "Piecewise Constant and Linear Regression Trees: An Optimal Dynamic Programming Approach," in *Proceedings of ICML*, 48994-49007 (2024).

[5] T. Huisman, **J.G.M. van der Linden**, and E. Demirović, "Optimal Survival Trees: A Dynamic Programming Approach," in *Proceedings of AAAI*, 12680-12688 (2024).

[6] **J.G.M. van der Linden**, M.M. de Weerdt, and E. Demirović, "Necessary and Sufficient Conditions for Optimal Decision Trees Using Dynamic Programming," in *Advances in NeurIPS*, 9173-9212 (2023).

[7] **J.G.M. van der Linden**, M.M. de Weerdt, and E. Demirović, "Fair and Optimal Decision Trees: A Dynamic Programming Approach," in *Advances in NeurIPS*, 38899-38911 (2022).

Other publications

[8] **J.G.M. van der Linden**, J. Mulderij, B. Huisman, J.W. den Ouden, M. van den Akker, H. Hoogeveen, and M.M. de Weerdt, "TORS: A Train Unit Shunting and Servicing Simulator," in *Proceedings of AAMAS*, 1773-1775 (2021).

[9] **K. van der Linden**, N. Romero, and M.M. de Weerdt, "Benchmarking Flexible Electric Loads Scheduling Algorithms," *Energies* 14(5), 1269 (2021).

[10] N. Romero, **K. van der Linden**, G. Morales-España, and M. de Weerdt, "Stochastic bidding of volume and price in constrained energy and reserve markets," *Electric Power Systems Research* 191, 106868 (2021).

[11] **K. van der Linden**, M. de Weerdt, and G. Morales-España, "Optimal Non-Zero Price Bids for EVs in Energy and Reserves Markets using Stochastic Optimization," in *Proceedings of the 15th International Conference on the European Energy Market* (2018).

[12] M. de Weerdt, M. Albert, V. Conitzer, and **K. van der Linden**, "Complexity of Scheduling Charging in the Smart Grid," in *Proceedings of IJCAI*, 4736-4742 (2018).

Preprints

[13] K. Sidorov, **K. van der Linden**, G.H.A. Correia, M.M. de Weerdt, and E. Demirović, "How to discover short, shorter, and the shortest proofs of unsatisfiability: a branch-and-bound approach for resolution proof length minimization," *arXiv preprint arXiv:2411.07955* (2024).

[14] J. Mulderij, B. Huisman, D. Tönissen, **K. van der Linden**, and M. de Weerdt, "Train unit shunting and servicing: A real-life application of multi-agent path finding," *arXiv preprint arXiv:2006.10422* (2020).