



The voice of optimization

Dimitris Bertsimas¹ · Bartolomeo Stellato¹ 

Received: 10 August 2019 / Revised: 2 June 2020 / Accepted: 13 June 2020 / Published online: 19 July 2020
© The Author(s), under exclusive licence to Springer Science+Business Media LLC, part of Springer Nature 2020

Abstract

We introduce the idea that using optimal classification trees (OCTs) and optimal classification trees with-hyperplanes (OCT-Hs), interpretable machine learning algorithms developed by Bertsimas and Dunn (Mach Learn 106(7):1039–1082, 2017), we are able to obtain insight on the strategy behind the optimal solution in continuous and mixed-integer convex optimization problem as a function of key parameters that affect the problem. In this way, optimization is not a black box anymore. Instead, we redefine optimization as a multiclass classification problem where the predictor gives insights on the logic behind the optimal solution. In other words, OCTs and OCT-Hs give optimization a voice. We show on several realistic examples that the accuracy behind our method is in the 90–100% range, while even when the predictions are not correct, the degree of suboptimality or infeasibility is very low. We compare optimal strategy predictions of OCTs and OCT-Hs and feedforward neural networks (NNs) and conclude that the performance of OCT-Hs and NNs is comparable. OCTs are somewhat weaker but often competitive. Therefore, our approach provides a novel insightful understanding of optimal strategies to solve a broad class of continuous and mixed-integer optimization problems.

Keywords Parametric optimization · Interpretability · Sampling · Multiclass classification

1 Introduction

Optimization has a long and distinguished history that has had and continues to have genuine impact in the world. In a typical optimization problem in the real world, practitioners see optimization as a black-box tool where they formulate the problem and they pass it to a solver to find an optimal solution. Especially in high dimensional problems typically encountered in real world applications, it is currently not possible to interpret or intuitively understand the optimal solution. However, independent from the optimization algorithm

Editor: Hendrik Blokkeel.

✉ Bartolomeo Stellato
stellato@mit.edu

Dimitris Bertsimas
dbertsim@mit.edu

¹ Operations Research Center and Sloan School of Management, Massachusetts Institute of Technology, Cambridge, MA 02139, USA

used, practitioners would like to understand how problem parameters affect the optimal decisions in order to get intuition and interpretability behind the optimal solution. Moreover, in almost all real world applications of optimization the main objective is not to solve just one problem but to solve multiple similar instances that vary slightly from each other. In fact, in most real-world applications we solve similar optimization problems multiple times with varying data depending on problem-specific parameters.

Our goal in this paper is to propose a framework to predict the optimal solution as parameters of the problem vary and do so in an interpretable way. A naive approach could be to learn directly the optimal solution from the problem parameters. However, this method is computationally *intractable* and *imprecise*: intractable, because it would require a potentially high dimensional predictor with hundreds of thousands of components depending on the decision variable size; imprecise, because it would involve a regression task that would naturally carry a generalization error leading to suboptimal or even infeasible solutions. Instead, our approach encodes the optimal solution with a small amount of information that we denote as *strategy*. Depending on the problem class, a strategy can have different meanings but it always corresponds to the complete information needed to efficiently recover the optimal solution.

In recent times, machine learning has also had significant impact to the world. Optimization methods has been a major driver of its success (Hastie et al. 2009; Bertsimas and Dunn 2019). In this paper, we apply machine learning to optimization with the objective to give a voice to optimization, that is to provide interpretability and intuition behind optimal solutions.

First, we solve an optimization problem for many parameter combinations and obtain the optimal strategy: the set of active constraints as well as the value of the discrete variables. Second, we encode the strategies with unique integer scalars. In this way, we have a mapping from parameters to the optimal solution strategies, which gives rise to a multiclass classification problem (Hastie et al. 2009). We solve the multiclass classification problem using OCTs and OCT-Hs developed by Bertsimas and Dunn (2017, 2019), interpretable state of the art classification algorithms as well as NNs, which, while not interpretable, serve as a benchmark to compare the accuracy of OCTs and OCT-Hs.

1.1 Related work

There has been a significant interest from both the computer science and the optimization communities to systematically analyze and solve optimization problems using machine learning. From the first works applying machine learning as a substitute for mathematical optimization algorithms in the 1990s (Smith 1999), this research direction has been increasingly active until recent results on learning for combinatorial optimization (Bengio et al. 2018).

1.1.1 Learning to tune algorithms

Most optimization methods consist of iterative routines with several parameters obtained through experts' knowledge and manual tuning. For example, Gurobi optimizer has 157 parameters (Gurobi Optimization 2020) that are carefully tuned in every release. However, this tuning can be, on the one hand, very complex because it concerns several aspects of the problem that are not known a priori and, on the other hand, suboptimal for the instances considered in a specific application. To overcome these limitations, the machine learning

community studied the *algorithm configuration* problem, i.e., the problem of automatically tuning algorithms from the problem instances of interest. One of the first efficient automatic configuration methods is ParamILS (Hutter et al. 2009) which iteratively improves the algorithm parameters using local search methods. Hutter et al. (2011) extended this idea in SMAC, a framework to learn the relationship between algorithm configurations and performance. The proposed scheme relies on Gaussian Processes and Random Forests to perform the predictions. Later, López-Ibáñez et al. (2016) introduced irace, an algorithm that iteratively refines the candidate parameters configurations to identify the best ones for a specific application. Despite their relevance in practice, these approaches aim to identify or predict the best algorithm parameters, independently from the algorithm ultimate task, i.e., they do not only consider optimization algorithms. In addition, they do not consider interpretability of the predictors. For instance, SMAC could benefit from interpretable predictors such as OCTs to allow the user to understand why some parameter configurations are better than others. Our work, instead, studies the relationship between the problem instances and the optimal solution focusing specifically on optimization algorithms and the interpretability of the optimal solutions.

1.1.2 Learning heuristics

Optimization methods not only rely on careful parameter tuning, but also on efficient heuristics. For example, branch-and-bound (B&B) involves several heuristic decisions about the branching behavior that are hand-tuned into the solvers. However, heuristics tuning can be very complex because it concerns several aspects of the problem that are not known a priori. Khalil et al. (2016) propose to learn the branching rules showing performance improvements over commercial hand-tuned algorithms. Similarly, Alvarez et al. (2017) approximate strong branching rules with learning methods. Machine learning has been useful also to select reformulations and decompositions for mixed-integer optimization (MIO). Bonami et al. (2018) learn in which cases it is more efficient to solve mixed-integer quadratic optimization problem (MIQO) by linearizing or not the cost function. They model it as a classification problem showing advantages compared to how this choice is made heuristically inside state-of-the-art solvers. Kruber et al. (2017) propose a similar method applied to decomposition selection for MIO.

1.1.3 Reinforcement learning for optimization

Another interesting line of research models optimization problems as control tasks to tackle using reinforcement learning (Sutton and Barto 2018). Problems suitable to this framework include knapsack-like or network problems with multistage decisions. Dai et al. (2017) develop a method to learn heuristics over graph problems. In this way the node selection criterion becomes the output of a specialized neural network that does not depend on the graph size (Dai et al. 2016). Every time a new node is visited, Dai et al. (2017) feed a graph representation of the problem to the NN obtaining a criterion suggesting the next node to select in the optimal solution.

1.1.4 Learning constraint programs

Constraint programming is a paradigm to model and solve combinatorial optimization problems very popular in the computer science community. The first works on applying

machine learning to automatically configure constraint programs (CPs) date back to the 1990s (Minton 1996). Later, Clarke et al. (2002) used Decision Trees to replace computationally hard parts of counterexample guided SAT solving algorithms. More recently, Xu et al. (2008) describe SATzilla, an automated approach for learning which candidate solvers are best on a given instance. SATzilla won several SAT solver competitions because of its ability to adapt and pick the best algorithm for a problem instance. Given a SAT instance, instead of solving it with different algorithms, SATzilla relies on a *empirical hardness model* to predict how long each algorithm should take. This model consists of a ridge regressor (Hastie et al. 2009) after nonlinear transformation of the problem features. Selsam et al. (2019) applied recent advances in NN architectures to CPs by directly predicting the solution or infeasibility. Even though this approach did not give as good results as state-of-the-art methods, it introduces a new research direction for solving CPs. Therefore, the constraint programming community is also working on data-driven methods to improve the performance and understanding of solution algorithms.

1.1.5 Learning parametric programs

Even though recent approaches for integrating machine learning and optimization show promising results, they do not consider the parametric nature of the problems appearing in real-world applications. It is often the case that practitioners solve the same problem with slightly varying parameters multiple times generating a large amount of data describing how the parameters affect the optimal solution. There are only a few recent papers exploiting this information to build better solution algorithms. The idea of learning the set of active constraints for parametric online optimization has been proposed by Misra et al. (2019). The authors frame the learning procedure as a sampling scheme where they collect all the active sets appearing from the parameters. However, we found several limitations of their approach. First, in the online phase, they evaluate all the relevant active sets in parallel and pick the best one (*ensemble policy* Misra et al. 2019). Therefore, they do not solve the optimization problem as a multiclass classification problem and they are not able to gain insights on how the parameters affect the optimal strategy. In addition, they do not tackle mixed-integer optimization problems but only continuous convex ones. Finally, the sampling strategy by Misra et al. (2019) has to be tuned for each specific problem since it depends on at least four different parameters. This is because the authors compute the probabilistic guarantees based on how many new active sets appear over the samples in a window of a specific size (Misra et al. 2019, Sect. 3). In this work, instead, we provide a concise Good–Turing estimator (Good 1953) for the probability of finding new unseen strategies which can be directly applied to many different problem instances. In the field of model predictive control (MPC), Klaučo et al. (2019) warm-start an online active set method for solving quadratic optimization problems (QOs). However, in that work there is no rigorous sampling scheme with probability bounds to obtain the different active sets. Instead, the authors either simulate the dynamical controlled system or provide an alternative gridding method to search for the relevant active sets. Furthermore, the method by Klaučo et al. (2019) is tailored to a specific linear control problem in the form of QO and cannot tackle general convex or mixed-integer convex problems. Learning for parametric programs can also speedup the online solution algorithms. Bertsimas and Stellato (2019) apply the framework in this paper to online mixed-integer optimization. By focusing on speed instead of interpretability, they obtain significant speedups compared to state-of-the-art solvers.

1.1.6 Sensitivity analysis

The study of how changes in the problem parameters affect the optimal solution has long been studied in sensitivity analysis, see Bertsimas and Tsitsiklis (1997, Chapter 5) and Boyd and Vandenberghe (2004, Sect. 5.6) for introductions on the topic. While sensitivity analysis is related to our work as it analyzes the effects of changes in problem parameters, it is fundamentally different both in philosophy and applicability. In sensitivity analysis the problem parameters are uncertain and the goal is to understand how their perturbations affect the optimal solution. This aspect is important when, for example, the problem parameters are not known with high accuracy and we would like to understand how the solution would change in case of perturbations. In this work instead, we consider problems without uncertainty and use previous data to learn how the problem parameters affect the optimal solution. Therefore, our problems are deterministic and we are not considering perturbations around any nominal value. As a matter of fact, the data we use for training are not restricted to lie close to any nominal point. In addition, sensitivity analysis usually studies continuous optimization problems since it relies on the dual variables at the optimal solution to determine the effect of parameter perturbations. This is why there has been only limited work on sensitivity analysis for MIO. In contrast, we show that our method can be directly applied to problems with integer variables.

1.2 Contributions

In this paper, we propose a learning framework to give a voice to continuous and mixed-integer convex optimization problems. With our approach we can reliably sample the occurring strategies using the Good–Turing estimator, learn an interpretable classifier using OCTs and OCT-Hs, interpret the dependency between the optimal strategies and the key parameters from the resulting tree and solve the optimization problem using the learned predictor.

Our specific contributions include:

1. We introduce a new framework for gaining insights on the solution of optimization problems as a function of their key parameters. The optimizer becomes an interpretable machine learning classifier using OCTs and OCT-Hs which highlights the relationship between the optimal solution strategy and the problem parameters. In this way optimization is no longer a black box and our approach gives it a voice that provides intuition and interpretability on the optimal solution.
2. We show that our method can be applied to a broad collection of optimization problems including convex continuous and convex mixed-integer optimization problem. We do not pose any assumption on the dependency of the cost and constraints on the problem parameters.
3. We introduce a new exploration scheme based on the Good–Turing estimator (Good 1953) to discover the strategies arising from the problem parameters. This scheme allows us to reliably bound the probability of encountering unseen strategies in order to be sure our classifier accurately represents the optimization problem.
4. In several realistic examples we show that the sample accuracy of our method is in the 90–100% range, while even in the cases where the prediction is not correct the degree of suboptimality or infeasibility is very low. We also compare the performance of OCTs

and OCT-Hs to NNs which can achieve state-of-the-art performance across a wide variety of prediction tasks. In our experiments we obtained comparable out-of-sample accuracy with OCTs, OCT-Hs and NNs.

In other words, our approach provides a novel, reliable and insightful framework for understanding the strategies to compute the optimal solution of a broad class of continuous and mixed-integer optimization problems.

1.3 Paper structure

The structure of the paper is as follows. In Sect. 2, we define an optimal strategy to solve continuous and mixed-integer optimization problems and present several concrete examples that demonstrate what we call the voice of optimization. In Sect. 3, we outline the core part of our approach of using multiclass classification to learn the mapping from parameters to optimal strategies. We further present an approach to estimate how likely it is to encounter a parameter that leads to an optimal strategy that has not yet been seen. In Sect. 4, we outline our Python implementation MLOPT (Machine Learning Optimizer). In Sect. 5, we test our approach on multiple examples from continuous and mixed-integer optimization and present the accuracy of predicting the optimal strategy of OCTs and OCT-Hs in comparison with a NNs implementation. Section 6 summarizes our conclusions. Appendix 1 and Appendix 2 briefly present optimal classification trees (OCTs and OCT-Hs) and NNs, respectively to make the paper self-contained.

2 The voice of optimization

In this section, we introduce the notion of an optimal strategy to solve continuous and mixed-integer optimization problems.

Given a parametric optimization problem, we define *strategy* $s(\theta)$ as the complete information needed to efficiently compute its optimal solution given the parameter $\theta \in \mathbf{R}^p$. We assume that the problem is always feasible for every encountered value of θ .

Note that in the illustrative examples in this section we omit the details of the learning algorithm which we explain in Sect. 3. Instead, we focus on the interpretation of the resulting strategies which correspond to a decision tree. For these examples the accuracy of our approach to find the optimal solution was always 100%, confirming that the resulting models accurately recover the optimal solution. For simplicity of exposition, we sample the problem parameters in the examples of this section in simple regions such as intervals or balls around specific points but this is not a strict requirement for our approach as it will be more clear in the next sections. In the examples in this section we used OCTs since their accuracy was 100% and they are more interpretable than OCT-Hs. In the last example, we used both an OCT and an OCT-H for comparison.

2.1 Optimal strategies in continuous optimization

Consider the continuous optimization problem

$$\begin{aligned} & \text{minimize} && f(\theta, x) \\ & \text{subject to} && g(\theta, x) \leq 0, \end{aligned} \quad (1)$$

where $x \in \mathbf{R}^n$ is the vector of decision variables and $\theta \in \mathbf{R}^p$ the vector of parameters affecting the problem data. Functions $f : \mathbf{R}^p \times \mathbf{R}^n \rightarrow \mathbf{R}$ and $g : \mathbf{R}^p \times \mathbf{R}^n \rightarrow \mathbf{R}^m$ are assumed to be convex in x . Given a parameter θ we denote the optimal primal solution as $x^*(\theta)$ and the optimal cost function value as $f(\theta, x^*(\theta))$.

2.1.1 Tight constraints

Let us define the *tight constraints* $\mathcal{T}(\theta)$ as the set of constraints that are satisfied as equalities at optimality,

$$\mathcal{T}(\theta) = \{i \in \{1, \dots, m\} \mid g_i(\theta, x^*) = 0\}. \quad (2)$$

Given the tight constraints, all the other constraints are no longer needed to solve the original problem.

For non-degenerate problems, the tight constraints correspond to the *support constraints*, i.e., the set of constraints that, if removed, would allow a decrease in $f(\theta, x^*(\theta))$ (Calafiore 2010, Definition 2.1). In the case of linear optimization problems (LOs) the support constraints are the linearly independent constraints defining a basic feasible solution (Bertsimas and Tsitsiklis 1997, Definition 2.9). An important property of support constraints is that they cannot be more than the dimension n of the decision variable (Hoffman 1979, Proposition 1), (Calafiore 2010, Lemma 2.2). This fact plays a key role in our method to reduce the complexity to predict the solution of parametric optimization problems. The benefits are more evident when the number of constraints is much larger than the number of variables, i.e., $n \ll m$.

2.1.2 Multiple optimal solutions and degeneracy

In practice we can encounter problems with multiple optimal solutions or degeneracy. When the cost function is not strongly convex in x , we can have multiple optimal solutions. In these cases we consider only one of the optimal solutions to be x^* since it is enough for our training purposes. Note that most solvers such as Gurobi Optimization (2020) return anyway only one solution and not the complete set of optimal solutions. With degenerate problems, we can have more tight constraints than support constraints for an optimal solution x^* . This is because the support constraints set is no longer unique in case of degeneracy. However, we use the set of tight constraints which remains unique since it includes by definition all the constraints that are satisfied as equalities independently from being support constraints or not. In addition, our method is still efficient because the number of tight constraints is in practice much lower than the total number of constraints, even in case of degeneracy. Therefore, we can directly apply our framework to problems with multiple optimal solutions and affected by degeneracy.

2.1.3 Solution strategy

We can now define our strategy as the index of tight constraints at the optimal solution, i.e., $s(\theta) = \mathcal{T}(\theta)$. Given the optimal strategy, solving (1) corresponds to solving

$$\begin{aligned} &\text{minimize} && f(\theta, x) \\ &\text{subject to} && g_i(\theta, x) \leq 0, \quad \forall i \in \mathcal{I}(\theta). \end{aligned} \tag{3}$$

This problem is easier than (1), especially when $n \ll m$. Note that we can enforce the components g_i that are linear in x as equalities. This further simplifies (3) while preserving its convexity. In case of LO and QO when the cost f is linear or quadratic and the constraints g are all linear, the solution of (3) corresponds to solving a linear system of equations defined by the KKT conditions (Boyd and Vandenberghe 2004, Sect. 10.2).

2.1.4 Inventory management

Consider an inventory management problem with horizon $t = 0, \dots, T - 1$ with $T = 30$. The decision variables are u_t , describing how much we order at time t and x_t , describing the inventory level at time t . The cost of ordering is $c = 2$, $h = 1$ is the holding cost and $p = 3$ is the shortage cost. We define the maximum quantity we can order each time as $M = 3$. The parameters are the product demand d_t at time t and the initial value of the inventory x_{init} . The optimization problem can be written as follows

$$\begin{aligned} &\text{minimize} && \sum_{t=0}^{T-1} \max(hx_t, -px_t) + cu_t \\ &\text{subject to} && x_{t+1} = x_t + u_t - d_t, \quad t = 0, \dots, T - 1 \\ &&& x_0 = x_{\text{init}} \\ &&& 0 \leq u_t \leq M, \quad t = 0, \dots, T - 1. \end{aligned} \tag{4}$$

Depending on d_t with $t = 0, \dots, T - 1$ and x_{init} , we need to adapt our ordering policy to minimize the cost. We assume that $d_t \in [1, 3]$ and $x_{\text{init}} \in [7, 13]$.

The strategy selection is summarized in Fig. 1 and can be easily interpreted: $u_t = 0$ for $t \leq t_0$ and then $u_t = d_t$ for $t > t_0$. We can explain this rule from the problem

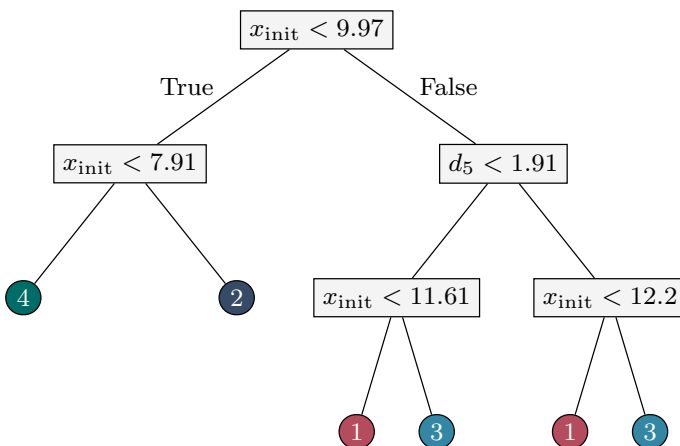


Fig. 1 Decision strategy for the inventory example. Strategy 1: do not order for the first 4 time steps, then order matching the demand. Strategy 2: do not order for the first 5 time steps, then order matching the demand. Strategy 3: do not order for the first 6 time steps, then order matching the demand. Strategy 4: do not order for the first 3 time steps, then order matching the demand

formulation. As discussed before, the strategy tells us which constraints are tight, in this case when $u_t = 0$ and, therefore, when we do not order. In the other time steps, we can ignore the inequality constraints and our goal is to minimize the cost by matching the demand. In this way, we do not have to store anything in the inventory, i.e., $x_t = 0$ and $\max(hx_t, -px_t) = 0$. Note that we anyway need to pay the cost of ordering cu_t since we have to satisfy the demand over the horizon.

An inventory level trajectory example appears in Fig. 2. Let us outline the strategies depicted in Fig. 1. For example, if the initial inventory level x_{init} is below 7.91, we should apply Strategy 4, where we wait only $t_0 = 3$ time steps before ordering. Otherwise, if $7.91 \leq x_{\text{init}} < 9.97$ we should wait for $t_0 = 5$ time steps before ordering because the initial inventory level is higher. The other branches can be similarly interpreted. Note that for this problem instance the decision is largely independent from d_t . The only exception comes with d_5 that determines the choice of strategies in the right-hand side of the tree.

Note that this is a simple illustrative example and the strategies shown are not all the theoretical ones. By allowing the problem parameters to take all the possible values we should expect a much larger number of strategies, and therefore a deeper tree. However, even though real-world examples can be more complicated, we very rarely hit all the possible strategy combinations.

The strategies we derived are related to the (s, S) policies for inventory management (Zheng and Federgruen 1991). If the inventory level x_t falls below value s , an (s, S) policy orders a quantity u_t so that the inventory is replenished to level S . This policy is remarkably simple because it consists only in a *if-then-else* rule. The optimal decisions from our approach are as simple as the (s, S) policies because we can describe them with the *if-then-else* rules as in Fig. 1. However, our method gives better performance because it takes into account future predictions and can deal with more complex problem formulations with harder constraints.

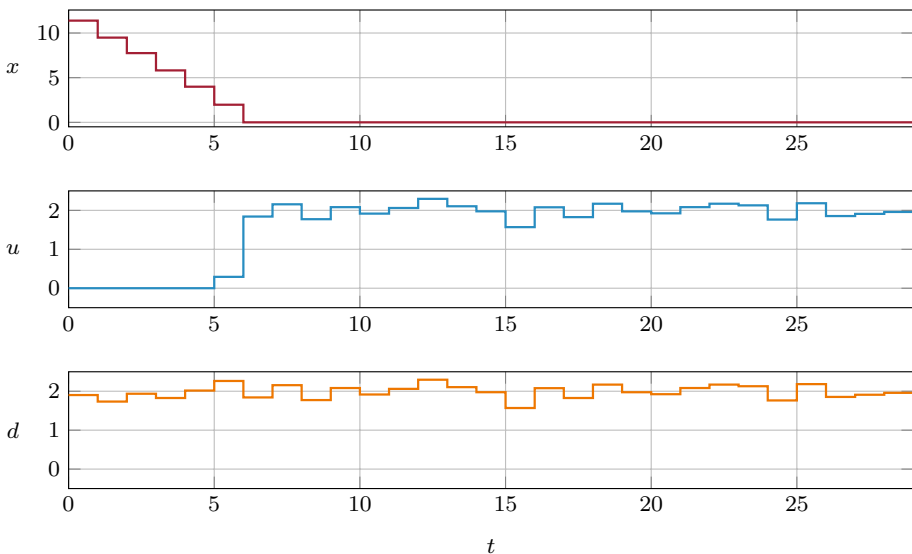


Fig. 2 Inventory behavior with Strategy 2. The lower bound on u_t is active for the first 5 steps. Then $u_t = d_t$

2.2 Optimal strategies in mixed-integer optimization

When dealing with integer variables we address the following problem

$$\begin{aligned} & \text{minimize} && f(\theta, x) \\ & \text{subject to} && g(\theta, x) \leq 0 \\ & && x_{\mathcal{I}} \in \mathbf{Z}^d. \end{aligned} \quad (5)$$

where \mathcal{I} is the set of indices for the variables constrained to be integer and $|\mathcal{I}| = d$.

2.2.1 Tight constraints

In this case the set of tight constraints does not uniquely define the optimal solution because of the integrality of some components of x . However, when we fix the integer variables to their optimal values $x_{\mathcal{I}}^*(\theta)$, (5) becomes a continuous convex optimization problem of the form

$$\begin{aligned} & \text{minimize} && f(\theta, x) \\ & \text{subject to} && g(\theta, x) \leq 0 \\ & && x_{\mathcal{I}} = x_{\mathcal{I}}^*(\theta). \end{aligned} \quad (6)$$

Note that the optimal cost of (6) and (5) are the same, however, optimal solutions may be different as there may be alternative optima. After fixing the integer variables, the tight constraints of problem (6) uniquely define an optimal solution.

2.2.2 Solution strategy

For this class of problems the strategy corresponds to a tuple containing the index of tight constraints of the continuous reformulation (6) and the optimal value of the integer variables, i.e., $s(\theta) = (\mathcal{T}(\theta), x_{\mathcal{I}}^*(\theta))$. Compared to continuous problems, we must also include the value of the integer variables to recover the optimal solution $x^*(\theta)$.

Given the optimal strategy, problem (5) corresponds to solving the continuous problem

$$\begin{aligned} & \text{minimize} && f(\theta, x) \\ & \text{subject to} && g_i(\theta, x) \leq 0, \quad \forall i \in \mathcal{T}(\theta) \\ & && x_{\mathcal{I}} = x_{\mathcal{I}}^*(\theta). \end{aligned} \quad (7)$$

Solving this problem is much less computationally demanding than (5) because it is continuous, convex and has smaller number of constraints, especially when $n \ll m$. As for the continuous case (3), the components g_i that are linear in x can be enforced as equalities further simplifying (7) while preserving its convexity.

Similarly to Sect. 2.1, in case of mixed-integer linear optimization problem (MILO) and MIQO when the cost f is linear or quadratic and the constraints g are all linear, the solution of (3) corresponds to solving a linear system of equations defined by the KKT conditions (Boyd and Vandenberghe 2004, Sect. 10.2). This means that we can solve

these problems online without needing any optimization solver (Bertsimas and Stellato 2019).

2.2.3 The knapsack problem

Consider the knapsack problem

$$\begin{aligned}
 &\text{maximize} && c^T x \\
 &\text{subject to} && a^T x \leq b \\
 & && 0 \leq x \leq u \\
 & && x \in \mathbf{Z}^n,
 \end{aligned} \tag{8}$$

with $n = 10$. The decision variables are $x = (x_1, \dots, x_{10})$ indicating the quantity to pick for each item $i = 1, \dots, 10$. We chose the cost vector $c = (0.42, 0.72, 0, 0.3, 0.15, 0.09, 0.19, 0.35, 0.4, 0.54)$. The knapsack capacity is $b = 5$. The weights $a = (a_1, \dots, a_{10})$ and the maximum order quantity $u = (u_1, \dots, u_{10})$ are our parameters. We assume that a is in a ball $B(a_0, r_0)$ and $u \in B(u_0, r_0)$ where $u_0 = a_0 = (2, 2, \dots, 2)$ and $r_0 = 1$.

With this setup we obtain the solution strategy outlined in Fig. 3. For this problem each strategy uniquely identifies the integer variables and is straightforward to analyze. For instance, the left part of the tree outlines what happens if the upper bound u_2 is strictly less than 2. Moreover, if the weight $a_1 < 1.4$, then it is easier to include x_1 in the knapsack solution and for this reason Strategy 3 has $x_1 = 2$ and $x_2 = 1$. On the contrary, if $a_1 \geq 1.4$, then Strategy 2 applies $x_1 = 0$ and $x_2 = x_{10} = 1$. Even though all these rules are simple, they capture the complexity of a hard combinatorial problem for the parameters that affect it. Note that only a few parameters affect the strategy selection, i.e., u_2, a_1 and a_2 , while the others are not relevant for this class of problem instances that have $a \in B(a_0, r_0)$ and $u \in B(u_0, r_0)$.

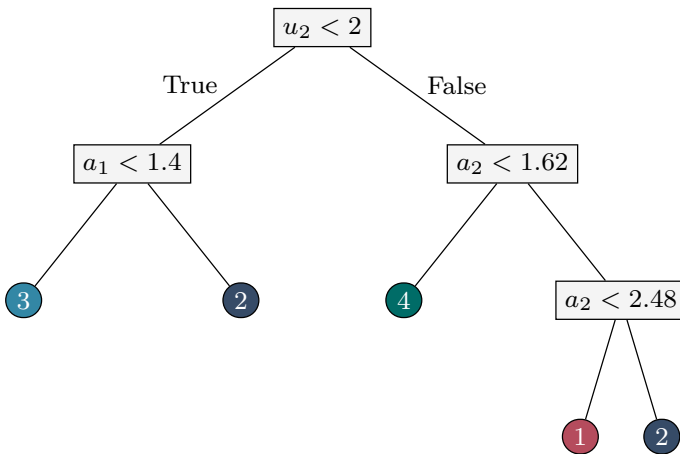


Fig. 3 Example knapsack decision strategies. Strategy 1: $x_i = 0$ for $i \neq 2$ and $x_2 = 2$. Strategy 2: $x_i = 0$ for $i \neq 2, 10$ and $x_2 = x_{10} = 1$. Strategy 3: $x_i = 0$ for $i \neq 1, 2$ and $x_1 = 2$ and $x_2 = 1$. Strategy 4: $x_i = 0$ for $i \neq 2, 10$ and $x_2 = 2$ and $x_{10} = 1$

2.2.4 Supplier selection

Consider the problem of selecting $n = 5$ suppliers in order to satisfy a known demand d . Our decision variables are $x \in \{0, 1\}^n$ determining which suppliers we choose and the amount of shipments u_i . For each supplier i we have a per-unit cost $c = (0.42, 0.72, 0, 0.3, 0.15)$ and a maximum quantity to order $m = (1.09, 1.19, 1.35, 1.4, 1.54)$, while $\gamma = 0.1$. We are interested in understanding the optimal strategy as a function of the demand d and the supplier delivery times τ_i , which are the problem parameters. The optimization problem is as follows:

$$\begin{aligned}
 &\text{minimize} && \sum_{i=1}^n c_i u_i + \gamma \max_i (\tau_i x_i) \\
 &\text{subject to} && \sum_{i=1}^n u_i \geq d \\
 &&& 0 \leq u_i \leq x_i m_i \\
 &&& x \in \{0, 1\}^n, u_i \in \mathbf{R}.
 \end{aligned} \tag{9}$$

Specifically, we are interested in $d \in [1, 3]$ and $\tau = (\tau_1, \dots, \tau_5) \in B(\tau_0, r_0)$ with the center of the ball being $\tau_0 = (2, 3, 2.5, 5, 1)$ and radius $r_0 = 0.5$. With these parameters we obtain the solution described in Fig. 4. Also in this case, the decision rules are simple and the solution strategy can be interpreted from the problem structure. The strategy outputs directly the optimal choice of suppliers x_i and which inequalities are tight corresponding to where we order the maximum quantity, i.e., when $u_i = m_i$. Note that the demand constraint is always tight by construction, i.e., we want to spend the minimum to match d . Therefore, we can fix x_i and u_i to the values given by the strategy and we can set the remaining u_i s to the minimum value such that $\sum_{i=1}^n u_i \geq d$.

In Fig. 5, we show the tree that OCT-H gave for this example. Even though the accuracy of the OCT in Fig. 4 is 100%, the OCT-H depth is smaller at the cost of being less

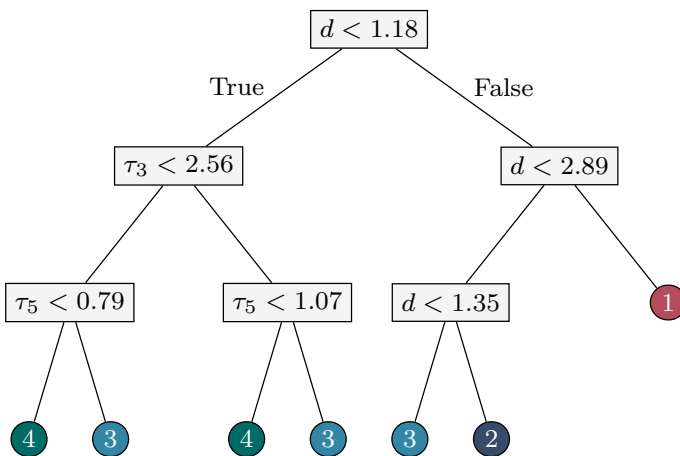


Fig. 4 Example vendor decision strategies using OCT. Strategy 1: $x = (1, 0, 1, 0, 1)$, $u_2 = u_4 = 0$. $u_3 = m_3, u_5 = m_5$ and u_1 is left to match demand d . Strategy 2: $x = (0, 0, 1, 0, 1)$. $u_1 = u_2 = u_4 = 0$ and $u_3 = m_3, u_5$ is left to match demand d . Strategy 3: $x = (0, 0, 1, 0, 0)$. $u_1 = u_2 = u_4 = u_5 = 0$ and u_3 is left to match demand d . Strategy 4: $x = (0, 0, 0, 0, 1)$. $u_1 = u_2 = u_3 = u_4 = 0$. u_5 is left to match demand d

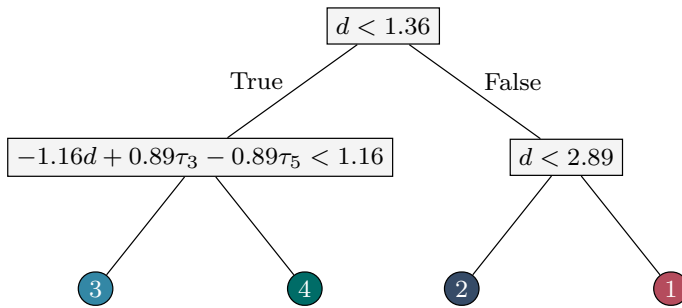


Fig. 5 Example vendor decision strategies using OCT-H. The strategies are the same as in Fig. 4 but accuracy is the same or higher and the tree depth is reduced. Smaller tree depth can sometimes help in interpreting the classification

interpretable. Note that in some cases having a lower depth can make some classification tasks more interpretable despite the multiple coefficients on the hyperplanes.

3 Machine learning

In this section, we introduce the core part of our approach: learning the mapping from parameters to optimal strategies. After the training, the mapping will replace the core part of standard optimization algorithms—the optimal strategy search—with a multiclass classification problem where each strategy corresponds to a class label.

3.1 Multiclass classifier

We would like to solve a multiclass classification problem with i.i.d. data (θ_i, s_i) , $i = 1, \dots, N$ where $\theta_i \in \mathbf{R}^p$ are the parameters and $s_i \in \mathcal{S}$ the corresponding labels identifying the optimal strategies. \mathcal{S} represents the set of strategies of cardinality $|\mathcal{S}| = M$.

In this work we apply two supervised learning techniques for multiclass classification to compare their predictive performance: optimal classification trees (OCTs, OCT-Hs) (Bertsimas and Dunn 2017, 2019) and neural networks (NNs) (Bengio 2009; LeCun et al. 2015). As we mentioned earlier OCTs, OCT-Hs are interpretable and can be described using simple rules as in the examples in the previous section, while NNs are not interpretable since they represent a composition of multiple nonlinear functions. A more detailed description of OCTs and OCT-Hs can be found in [Appendix 1](#) and of NNs can be found in [Appendix 2](#).

3.2 Strategies exploration

In this section, we estimate how likely it is to find a new parameter θ whose optimal strategy does not lie among the ones we have already seen. If we have already encountered most of the strategies for our problem, then it is unlikely to find unseen strategies. Therefore, we can be sure that our classification problem includes all the possible strategies

(classes) arising in practice. Otherwise, we must collect more data to have a more representative classification problem.

3.2.1 Estimating the probability of finding unseen strategies

Given N independent samples $\Theta_N = \{\theta_1, \dots, \theta_N\}$ drawn from an unknown discrete distribution \mathcal{P} with the corresponding strategies $(s(\theta_1), \dots, s(\theta_N))$, we find M unique strategies $\mathcal{S}(\Theta_N) = \{s_1, \dots, s_M\}$. We are interested in bounding the probability of finding unseen strategies

$$\mathbf{P}(s(\theta_{N+1}) \notin \mathcal{S}(\Theta_N)),$$

with confidence at least $1 - \beta$ where $\beta > 0$.

3.2.2 Historical background

This problem started from the seminal work by Turing and Good (Good 1953) in the context of decrypting the Enigma codes during World War II. The Enigma machine was a German navy encryption device used for secret military communications. Part of Enigma's encryption key was a three letter sequence (a word) selected from a book containing all the possible ones in random order. The number of possible words was so large that it was impossible to test all the combinations with the computing power available at that time. In order to decrypt the Enigma machine without testing all the possible words, Turing wanted to check only a subset of them while estimating that the likelihood of finding a new unseen word was low. This is how the Good–Turing estimator was developed. It was a fundamental step towards the Enigma machine decryption which is believed to have shortened World War II of at least 2 years (Copeland 2012). In addition, this class of estimators have become standard in a wide range of natural language processing applications.

3.2.3 Good–Turing estimator

Let N_r be the number of strategies that appeared exactly r times in $(s(\theta_1), \dots, s(\theta_N))$. The Good–Turing estimator for the probability of having an unseen strategy is given by Good (1953)

$$G = N_1/N, \quad (10)$$

which corresponds to the ratio between the number of distinct strategies that have appeared exactly once, over the total number of samples. Despite the elegant result, Good and Turing did not provide a convergence analysis of this estimator for finite samples. Only few decades later the first theoretical work on that topic appeared in McAllester and Schapire (2000). Using McDiarmid (1989)'s inequality the authors derived a high probability confidence interval. That result can be directly applied to our problem with the following theorem.

Theorem 3.1 (Missing strategies bound) *The probability of encountering a parameter θ_{N+1} corresponding to an unseen strategy $s(\theta_{N+1})$ satisfies with confidence at least $1 - \beta$*

$$\mathbf{P}(s(\theta_{N+1}) \notin \mathcal{S}(\Theta_N)) \leq G + c\sqrt{(1/N)\ln(3/\beta)},$$

where G corresponds to the Good–Turing estimator (10) and $c = (2\sqrt{2} + \sqrt{3})$.

Proof of Theorem 3.1 The result follows directly from McAllester and Schapire (2000, Theorem 9). \square

3.2.4 Exploration algorithm

Given the bound in Theorem 3.1 we construct Algorithm 1 to compute the different strategies appearing in our problem. The algorithm keeps on sampling until we encounter a large enough set of distinct strategies. It terminates when the probability of encountering a parameter with an unseen optimal strategy is less than ϵ with confidence at least $1 - \beta$. Note that in practice, instead of sampling one point per iteration k , it is more convenient to sample more points to avoid having to recompute the class frequencies which becomes computationally intensive with several thousands of samples and hundreds of classes. Algorithm 1 imposes no assumptions on the optimization problem nor the data distribution apart from i.i.d. samples.

Algorithm 1 Strategies exploration

```

1: given  $\epsilon, \beta, \Theta = \emptyset, \mathcal{S} = \emptyset, u = \infty$ 
2: for  $k = 1, \dots$ , do
3:   Sample  $\theta_k$  and compute  $s(\theta_k)$  ▷ Sample parameter and strategy.
4:    $\Theta \leftarrow \Theta \cup \{\theta_k\}$  ▷ Update set of samples.
5:   if  $s(\theta_k) \notin \mathcal{S}$  then
6:      $\mathcal{S} \leftarrow \mathcal{S} \cup \{s(\theta_k)\}$  ▷ Update strategy set if new strategy found
7:     if  $G + c\sqrt{(1/k) \ln(3/\beta)} \leq \epsilon$  then ▷ Break if bound less than  $\epsilon$ 
8:       break
9: return  $k, \Theta, \mathcal{S}$ 

```

4 Machine learning optimizer

We implemented our method in the Python software tool MLOPT (Machine Learning Optimizer). It is integrated with CVXPY (Diamond and Boyd 2016) to formulate and solve the problems. CVXPY allows the user to easily define mixed-integer convex optimization problems performing all the required reformulations needed by the optimizers while keeping track of the original constraints and variables. This makes it ideal for identifying which constraints are tight or not at the optimal solution. The MLOPT implementation is freely available online at

github.com/bstellato/mlopt.

We used the Gurobi Optimizer (Gurobi Optimization 2020) to find the tight constraints because it provides a good tradeoff between solution accuracy and computation time. Note that from Sect. 2, in case of LO, MILO, QO and MIQO when the cost f is linear or quadratic and the constraints g are all linear, the online solution corresponds to solving a linear system of equations defined by the KKT conditions (Boyd and Vandenberghe 2004,

Sect. 10.2) on the reduced subproblem. This means that we can solve those parametric optimization problems without the need to apply any optimization solver.

MLOPT performs the iterative sampling procedure outlined in Sect. 3.2 to obtain the strategies required for the classification task. We sample 5000 new points at each iteration and compute their strategies until the Good Turing estimate is below $\epsilon_{GT} = 0.005$. The strategy computation is fully parallelized across samples.

We interfaced MLOPT to the machine learning libraries OptimalTrees.jl (Bertsimas and Dunn 2019) on multi-core CPUs and PyTorch (Paszke et al. 2017) for both CPUs and GPUs. In the training phase we automatically tune the predictor parameters by splitting the data points in 90/10% training/validation. We tune the OCTs and OCT-Hs with maximal depth for values 5, 10, 15 and minimum bucket size for values 1, 5, 10. For the NNs we validate the stochastic gradient descent learning rate for values 0.001, 0.01, 0.1, batch size for values 32, 128 and number of epochs for values 50, 100. We described the complete algorithm in Fig. 6.

5 Computational benchmarks

In this section, we test our approach on multiple examples from continuous and mixed-integer optimization. We benchmarked the predictive performance of OCTs, OCT-Hs (see Appendix 1) and NNs (see Appendix 2) depending on the problem type and size. We executed the numerical tests on a Dell R730 cluster with 28 Intel E5-2680 CPUs with a total of 256 GB RAM and a Nvidia Tesla K80 GPU. To facilitate the data collection, we generated the training samples from distributions on intervals or on hyperballs around specified points. In this way the distributions are unimodal and therefore closer to the ones encountered in practice. Note that our theoretical results do not assume anything on the distribution generating data apart from i.i.d. samples. Therefore, we could have chosen other distributions such as multimodal ones. In general, how to generate training data highly depends on the practical application and in many cases we can simply fit a representative distribution, either unimodal or multimodal, to the historical data we have. We evaluated the performance metrics on 100 unseen samples drawn from the same distribution of θ .

5.1 Infeasibility and suboptimality

After the learning phase, we compared the predicted solution \hat{x}_i^* to the optimal one x_i^* obtained by solving the instances from scratch. Given a parameter θ_i , we say that the

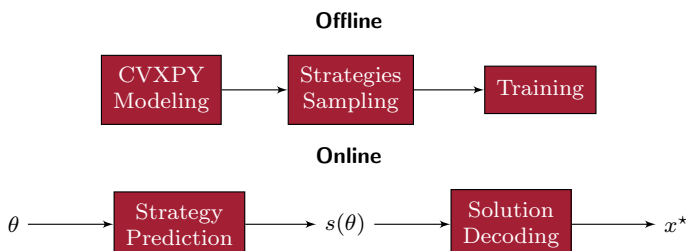


Fig. 6 Algorithm implementation

predicted solution is infeasible if the constraints are violated more than a predefined tolerance $\epsilon_{\text{inf}} = 10^{-3}$ according to the infeasibility metric

$$p(\hat{x}_i^*) = \|(g(\theta_i, \hat{x}_i^*))_+\|_2 / r(\theta_i, \hat{x}_i^*),$$

where $r(\theta, x)$ normalizes the violation depending on the size of the summands of g . For example, if $g(\theta, x) = A(\theta)x - b$, then $r(\theta, x) = \max(\|A(\theta)x\|_2, \|b\|_2)$. If the predicted solution \hat{x}_i is feasible, we define its suboptimality as

$$d(\hat{x}_i^*) = (f(\hat{x}_i^*) - f(x_i^*)) / |f(x_i^*)|.$$

Note that $d(\hat{x}_i^*) \geq 0$ by construction. We consider a predicted solution to be accurate if it is feasible and if the suboptimality is less than the tolerance $\epsilon_{\text{sub}} = 10^{-3}$. For each instance we report the maximum infeasibility $\bar{p} = \max_i p(\hat{x}_i^*)$ and the maximum suboptimality $\bar{d} = \max_i d(\hat{x}_i^*)$ over the test dataset. Note that for notation ease and to simplify the max operation, if a point is infeasible we consider its suboptimality 0 and ignore it.

5.2 Predicting the optimal strategy

Once the learning phase is completed, the predictor outputs the three (3) most likely optimal strategies and picks the best one according to infeasibility and suboptimality after solving the associated reduced problems.

5.3 Runtimes

The training times of these examples range from a 2 to 12 h including the time to solve the problems in parallel over the training phase and the time to train the predictors. Even though this time can be significant, it is comparable to the usual time dedicated to train predictors for modern machine learning tasks. We report the time ratio t_{ratio} between the time needed to solve the problem with Gurobi (Gurobi Optimization 2020) compared to our method.

5.4 Tables notation

We now outline the table headings as reported in every example. Dimensions n and/or m denote problem-specific sizes that vary over the benchmarks. The total number of variables and constraints are n_{var} and n_{con} respectively. Column “learner” indicates the learner type: OCT, OCT-H or NN. We indicate the Good–Turing estimator under column “GT” and the accuracy over the test set as “acc [%]”. The column t_{ratio} denotes the speedup of our method over Gurobi (Gurobi Optimization 2020). The maximum infeasibility appears under column \bar{p} and the maximum suboptimality under column \bar{d} .

5.5 Transportation optimization

Consider a standard transportation problem with n warehouses and m retail stores. Let x_{ij} be the quantity we ship from warehouse i to store j . s_i denotes the supply for warehouse i and d_j the demand from store j . We define the cost of transporting a unit from warehouse i to store j as c_{ij} . The optimization problem can be formulated as follows:

$$\begin{aligned}
 &\text{minimize} && \sum_{i=1}^n \sum_{j=1}^m c_{ij}x_{ij} \\
 &\text{subject to} && \sum_{j=1}^m x_{ij} \leq s_i, && \forall i \\
 &&& \sum_{i=1}^n x_{ij} \geq d_j && \forall j \\
 &&& x_{ij} \geq 0 && \forall i, j.
 \end{aligned}$$

The first constraint ensures we respect the supply for each warehouse i . The second constraint enforces the sum of all the shipments to match at least the demand for store j . Finally, the quantity of shipped product has to be nonnegative. Our learning parameters are the demands d_j .

5.6 Problem instances

We generate the transportation problem instances by varying the number of warehouses n and stores m . The cost vectors c_i are distributed as $\mathcal{U}(0, 5)$ and the supplies s_i as $\mathcal{U}(3, 13)$. The parameter vector $d = (d_1, \dots, d_m)$ was sampled from a uniform distribution within the ball $B(\bar{d}, 0.75)$ with center $\bar{d} \sim \mathcal{N}(3, 1)$.

5.7 Results

The results appear in Table 1. Independently from the problem instances the prediction accuracy is very high for both optimal trees and neural networks. Note that even though we solve problems with thousands of variables and constraints, the number of strategies is always within few tens or less. The solution time provides usually slight speedups up to almost 5 folds. Note that in the very small instances Gurobi can be faster than our method due to the delays in data exchange between predictor and our Python code.

5.8 Portfolio optimization

Consider the problem of allocating assets to minimize the risk adjusted return considered in Markowitz (1952) who formulated it as a QO

$$\begin{aligned}
 &\text{maximize} && \mu^T x - \gamma(x^T \Sigma x) \\
 &\text{subject to} && \mathbf{1}^T x = 1 \\
 &&& x \geq 0,
 \end{aligned}$$

where the variable $x \in \mathbf{R}^n$ represents the investments to make.

Our learning parameter is the vector of expected returns $\mu \in \mathbf{R}^n$. In addition we denote the risk aversion coefficient as $\gamma > 0$, and the covariance matrix for the risk model as $\Sigma \in \mathbf{S}_+^n$. Σ is assumed to be

$$\Sigma = FF^T + D,$$

where $F \in \mathbf{R}^{n \times p}$ is the factor loading matrix and $D \in \mathbf{R}^{n \times n}$ is a diagonal matrix describing the asset-specific risk.

Table 1 Transportation benchmarks

n	m	n_{var}	n_{con}	Learner	N	GT	$ \mathcal{S} $	acc (%)	t_{ratio}	\bar{p}	\bar{d}
20	20	400	440	NN	10,000	1.00×10^{-4}	11	100.00	0.81	6.34×10^{-5}	4.40×10^{-16}
20	20	400	440	OCT	10,000	1.00×10^{-4}	11	100.00	0.04	6.34×10^{-5}	4.40×10^{-16}
20	20	400	440	OCT-H	10,000	1.00×10^{-4}	11	100.00	1.22	6.34×10^{-5}	4.40×10^{-16}
20	10	200	230	NN	10,000	1.00×10^{-4}	6	100.00	0.74	1.42×10^{-16}	2.76×10^{-16}
20	10	200	230	OCT	10,000	1.00×10^{-4}	6	100.00	0.88	1.42×10^{-16}	2.76×10^{-16}
20	10	200	230	OCT-H	10,000	1.00×10^{-4}	6	100.00	1.36	1.42×10^{-16}	2.76×10^{-16}
40	40	1600	1680	NN	10,000	2.00×10^{-4}	9	100.00	1.95	3.00×10^{-16}	1.98×10^{-16}
40	40	1600	1680	OCT	10,000	2.00×10^{-4}	9	100.00	1.74	3.00×10^{-16}	1.98×10^{-16}
40	40	1600	1680	OCT-H	10,000	2.00×10^{-4}	9	100.00	1.81	3.00×10^{-16}	1.98×10^{-16}
40	20	800	860	NN	10,000	0.00	3	100.00	4.98	1.41×10^{-16}	2.11×10^{-16}
40	20	800	860	OCT	10,000	0.00	3	100.00	3.53	1.41×10^{-16}	2.11×10^{-16}
40	20	800	860	OCT-H	10,000	0.00	3	100.00	3.74	1.41×10^{-16}	2.11×10^{-16}
60	60	3600	3720	NN	10,000	1.50×10^{-3}	43	99.00	2.04	1.86×10^{-2}	2.46×10^{-16}
60	60	3600	3720	OCT	10,000	1.50×10^{-3}	43	95.00	1.80	2.72×10^{-2}	1.55×10^{-4}
60	60	3600	3720	OCT-H	10,000	1.50×10^{-3}	43	100.00	2.11	5.17×10^{-5}	2.46×10^{-16}
60	30	1800	1890	NN	10,000	8.00×10^{-4}	30	99.00	1.70	1.60×10^{-3}	1.97×10^{-16}
60	30	1800	1890	OCT	10,000	8.00×10^{-4}	30	100.00	1.48	1.19×10^{-16}	1.97×10^{-16}
60	30	1800	1890	OCT-H	10,000	8.00×10^{-4}	30	100.00	1.55	1.19×10^{-16}	1.97×10^{-16}
80	80	6400	6560	NN	10,000	1.10×10^{-3}	36	100.00	2.04	1.99×10^{-16}	2.20×10^{-16}
80	80	6400	6560	OCT	10,000	1.10×10^{-3}	36	100.00	2.12	1.99×10^{-16}	2.20×10^{-16}
80	80	6400	6560	OCT-H	10,000	1.10×10^{-3}	36	100.00	2.43	1.99×10^{-16}	2.20×10^{-16}
80	40	3200	3320	NN	10,000	1.00×10^{-4}	14	100.00	2.40	2.10×10^{-16}	3.23×10^{-16}
80	40	3200	3320	OCT	10,000	1.00×10^{-4}	14	100.00	2.13	2.10×10^{-16}	3.23×10^{-16}
80	40	3200	3320	OCT-H	10,000	1.00×10^{-4}	14	100.00	2.38	2.10×10^{-16}	3.23×10^{-16}

5.9 Problem instances

We generated portfolio instances for different number of factors p and assets n . We chose the elements of F with 50% nonzeros and $F_{ij} \sim \mathcal{N}(0, 1)$. The diagonal matrix D has elements $D_{ii} \sim \mathcal{U}(0, \sqrt{p})$. The return vector parameters were randomly sampled from a uniform distribution within the ball $B(\bar{\mu}, 0.15)$ with $\bar{\mu} \sim \mathcal{N}(0, 1)$. The risk-aversion coefficient is $\gamma = 1$.

5.10 Results

Results are shown in Table 2. The prediction accuracy for OCTs, OCT-Hs and NNs is 100%, the number of strategies is less than 15, and the infeasibility and suboptimality are very low. The solution times improvement is not significant due to the delays of passing data to the predictor and the fact that Gurobi is fast for these problems.

Table 2 Continuous portfolio benchmarks

n	p	n_{con}	Learner	N	GT	$ \mathcal{S} $	acc (%)	t_{ratio}	\bar{p}	\bar{d}
100	10	101	NN	10,000	4.00×10^{-4}	12	100.00	1.01	6.66×10^{-16}	7.06×10^{-16}
100	10	101	OCT	10,000	4.00×10^{-4}	12	100.00	0.09	6.66×10^{-16}	7.06×10^{-16}
100	10	101	OCT-H	10,000	4.00×10^{-4}	12	100.00	1.07	6.66×10^{-16}	7.06×10^{-16}
200	20	201	NN	10,000	4.00×10^{-4}	11	100.00	0.88	4.44×10^{-16}	5.54×10^{-16}
200	20	201	OCT	10,000	4.00×10^{-4}	11	100.00	0.81	4.44×10^{-16}	3.84×10^{-7}
200	20	201	OCT-H	10,000	4.00×10^{-4}	11	100.00	0.79	4.44×10^{-16}	5.54×10^{-16}
300	30	301	NN	10,000	1.00×10^{-4}	4	100.00	2.87	3.38×10^{-5}	6.96×10^{-16}
300	30	301	OCT	10,000	1.00×10^{-4}	4	100.00	2.55	3.38×10^{-5}	6.96×10^{-16}
300	30	301	OCT-H	10,000	1.00×10^{-4}	4	100.00	2.46	3.38×10^{-5}	6.96×10^{-16}
400	40	401	NN	10,000	0.00	14	100.00	1.29	1.48×10^{-5}	5.26×10^{-16}
400	40	401	OCT	10,000	0.00	14	100.00	0.72	1.48×10^{-5}	5.26×10^{-16}
400	40	401	OCT-H	10,000	0.00	14	100.00	1.20	1.48×10^{-5}	1.82×10^{-7}

5.11 Facility location

Let $i \in I$ the set of possible locations for facilities such as factories or warehouses. We denote as $j \in J$ the set of delivery locations. The cost of transporting a unit of goods from facility i to location j is c_{ij} . The construction cost for building facility i is f_i . For each location j , we define the demand d_j . The capacity of facility i is s_i . The main goal of this problem is to find the best tradeoff between transportation costs versus construction costs. We can write the model as a MILO

$$\begin{aligned}
 & \text{minimize} && \sum_{i \in I} \sum_{j \in J} c_{ij} x_{ij} + \sum_{i \in I} f_i y_i \\
 & \text{subject to} && \sum_{i \in I} x_{ij} \geq d_j, \quad \forall j \in J \\
 & && \sum_{j \in J} x_{ij} \leq s_i y_i, \quad \forall i \in I \\
 & && x_{ij} \geq 0 \\
 & && y_i \in \{0, 1\}.
 \end{aligned} \tag{11}$$

The decision variables x_{ij} describe the amount of goods sent from facility i to location j . The binary variables y_i determine if we build facility i or not.

5.12 Problem instances

We generated the facility location instances for different values of facilities n and warehouses m . The costs c_{ij} are sampled from a uniform distribution $\mathcal{U}(0, 1)$ and f_i from $\mathcal{U}(0, 10)$. We chose capacities s_i as $\mathcal{U}(8, 18)$ to ensure the problem is always feasible. The demands parameters $\bar{d} = (d_1, \dots, d_m)$ were sampled from a uniform distributions within the ball $B(\bar{d}, 0.25)$ with $\bar{d} \sim \mathcal{N}(3, 1)$.

5.13 Results

Table 3 shows the benchmark examples. All prediction methods showed very high accuracy. There is a significant improvement in computation time compared to Gurobi. To illustrate an example of the OCT interpretability, we reported the resulting tree for $n = 60$ and $m = 30$ in Fig. 7. Even though for space concerns we cannot directly report the strategies in terms of tight constraints and integer variables since the problem involves 1860 variables and 2010 constraints, the tree is remarkably simple. In every strategy all the demand constraints are tight since we are trying to minimize the cost while satisfying the demand. In addition, the strategy illustrates which facilities hit the maximum capacity s_i which could mean that further facilities might be needed in that area. Finally, with the values of binary variables y_i , the strategy tells us which facilities we should open depending on the demand.

5.14 Hybrid vehicle control

Consider the hybrid-vehicle control problem in Takapoui et al. (2017, Sect. 3.2). The model consists of a battery, an electric motor/generator and an engine. We assume to know the demanded power P_t^{des} over the horizon $t = 0, \dots, T - 1$. The goal is to plan

Table 3 Facility location benchmarks

n	m	n_{var}	n_{con}	Learner	N	GT	$ S $	acc (%)	t_{ratio}	\bar{p}	\bar{d}
20	10	220	270	NN	10,000	0.00	1	100.00	537.67	1.06×10^{-16}	1.64×10^{-16}
20	10	220	270	OCT	10,000	0.00	1	100.00	3.87	1.06×10^{-16}	1.64×10^{-16}
20	10	220	270	OCT-H	10,000	0.00	1	100.00	573.24	1.06×10^{-16}	1.64×10^{-16}
20	20	420	480	NN	10,000	1.00×10^{-4}	9	100.00	149.06	9.45×10^{-5}	1.81×10^{-9}
20	20	420	480	OCT	10,000	1.00×10^{-4}	9	100.00	174.62	9.45×10^{-5}	1.81×10^{-9}
20	20	420	480	OCT-H	10,000	1.00×10^{-4}	9	100.00	234.11	9.45×10^{-5}	1.81×10^{-9}
40	20	840	940	NN	10,000	0.00	2	100.00	1986.00	1.76×10^{-16}	2.43×10^{-16}
40	20	840	940	OCT	10,000	0.00	2	100.00	2080.12	1.76×10^{-16}	2.43×10^{-16}
40	20	840	940	OCT-H	10,000	0.00	2	100.00	2201.23	1.76×10^{-16}	2.43×10^{-16}
40	40	1640	1760	NN	10,000	1.00×10^{-4}	5	100.00	1037.03	1.42×10^{-16}	2.50×10^{-9}
40	40	1640	1760	OCT	10,000	1.00×10^{-4}	5	100.00	961.91	1.42×10^{-16}	2.50×10^{-9}
40	40	1640	1760	OCT-H	10,000	1.00×10^{-4}	5	100.00	1070.20	1.42×10^{-16}	2.50×10^{-9}
60	30	1860	2010	NN	10,000	1.00×10^{-4}	3	100.00	325.76	2.02×10^{-16}	1.91×10^{-16}
60	30	1860	2010	OCT	10,000	1.00×10^{-4}	3	100.00	334.58	2.02×10^{-16}	1.91×10^{-16}
60	30	1860	2010	OCT-H	10,000	1.00×10^{-4}	3	100.00	296.36	2.02×10^{-16}	1.91×10^{-16}
60	60	3660	3840	NN	10,000	1.00×10^{-4}	5	100.00	341.24	3.01×10^{-16}	2.63×10^{-16}
60	60	3660	3840	OCT	10,000	1.00×10^{-4}	5	100.00	415.54	3.01×10^{-16}	2.63×10^{-16}
60	60	3660	3840	OCT-H	10,000	1.00×10^{-4}	5	100.00	409.08	3.01×10^{-16}	2.63×10^{-16}
80	40	3280	3480	NN	10,000	1.00×10^{-4}	9	100.00	97.38	3.03×10^{-16}	2.77×10^{-16}
80	40	3280	3480	OCT	10,000	1.00×10^{-4}	9	100.00	99.44	3.03×10^{-16}	2.77×10^{-16}
80	40	3280	3480	OCT-H	10,000	1.00×10^{-4}	9	100.00	89.52	3.03×10^{-16}	2.77×10^{-16}
80	80	6480	6720	NN	10,000	1.00×10^{-4}	6	100.00	865.45	3.34×10^{-16}	2.52×10^{-10}
80	80	6480	6720	OCT	10,000	1.00×10^{-4}	6	100.00	64.74	3.34×10^{-16}	2.52×10^{-10}
80	80	6480	6720	OCT-H	10,000	1.00×10^{-4}	6	100.00	1061.22	3.34×10^{-16}	2.52×10^{-10}

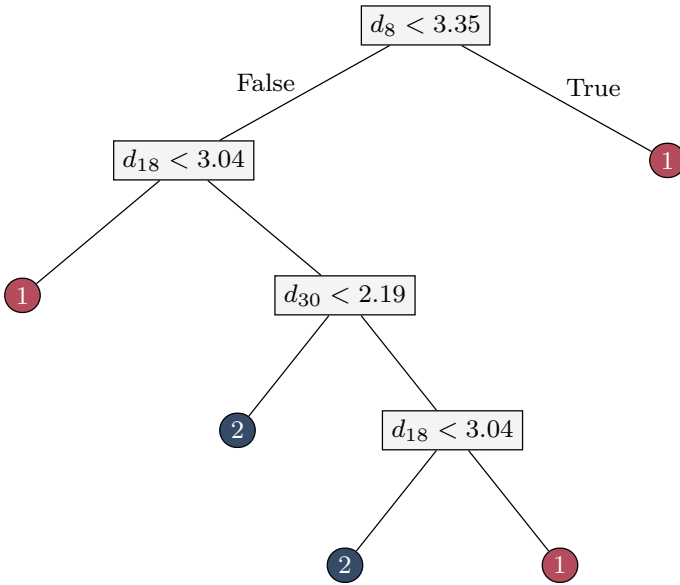


Fig. 7 Facility location benchmark OCT for $n = 60$ and $m = 30$

the battery and the engine power outputs P_t^{batt} and P_t^{eng} so that they match at least the demand,

$$P_t^{\text{batt}} + P_t^{\text{eng}} \geq P_t^{\text{des}}.$$

The status of the battery is modeled with the internal energy E_t evolving as

$$E_{t+1} = E_t - \tau P_t^{\text{batt}},$$

where $\tau > 0$ is the time interval discretization. The battery capacity is limited by E^{max} and its initial value is E_{init} .

We now model the cost function. We penalize the terminal energy state of the battery with the function

$$g(E) = \eta(E^{\text{max}} - E)^2.$$

with $\eta \geq 0$. At each time t we model the on-off state of the engine with the binary variable z_t . When the engine is off ($z_t = 0$) we do not consume any energy, thus we have $0 \leq P_t^{\text{eng}} \leq P^{\text{max}} z_t$. When the engine is on ($z_t = 1$) it consumes $\alpha P_t^{\text{eng}} + \beta P_t^{\text{eng}} + \gamma$ units of fuel, with $\alpha, \beta, \gamma > 0$. We define the stage power cost as

$$f(P, z) = \alpha P^2 + \beta P + \gamma z.$$

We also introduce a cost of turning the engine on at time t as $\delta(z_t - z_{t-1})_+$.

The hybrid vehicle control problem can be formulated as a mixed integer quadratic optimization problem:

$$\begin{aligned}
& \text{minimize} && \eta(E_T - E^{\max})^2 + \sum_{t=0}^{T-1} f(P_t^{\text{eng}}, z_t) + \delta(z_t - z_{t-1})_+ \\
& \text{subject to} && E_{t+1} = E_t - \tau P_t^{\text{batt}}, \quad t = 0, \dots, T-1 \\
& && 0 \leq E_t \leq E^{\max}, \quad t = 0, \dots, T \\
& && E_0 = E_{\text{init}} \\
& && 0 \leq P_t^{\text{eng}} \leq P^{\max}, \quad t = 0, \dots, T-1 \\
& && P_t^{\text{batt}} + P_t^{\text{eng}} \geq P_t^{\text{des}}, \quad t = 0, \dots, T-1 \\
& && z_t \in \{0, 1\}, \quad t = 0, \dots, T-1.
\end{aligned}$$

The initial state E_0 and demanded power P_t^{des} are our parameters.

5.15 Problem instances

We generated the control instances with varying horizon length T . The discretization time interval is $\tau = 4$. We chose the cost function parameters $\alpha = \beta = \gamma = 1$, and $\delta = 0.1$. The maximum charge is $E^{\max} = 50$ and the maximum power $P^{\max} = 1$. The parameter E_0 was sampled from a uniform distribution within the ball $B(40, 0.5)$. The demand P^{des} also comes from a uniform distribution within the ball $B(\bar{P}^{\text{des}}, 0.5)$ where

$$\begin{aligned}
\bar{P}_t^{\text{des}} = & (0.05, 0.30, 0.55, 0.80, 1.05, 1.30, 1.55, 1.80, 1.95, 1.70, 1.45, 1.20, 1.02, \\
& 1.12, 1.22, 1.32, 1.42, 1.52, 1.62, 1.72, 1.73, 1.38, 1.03, 0.68, 0.33, -0.02, \\
& -0.37, -0.72, -0.94, -0.64, -0.34, -0.04, 0.18, 0.08, -0.02, -0.12, \\
& -0.22, -0.32, -0.42, -0.52)
\end{aligned}$$

is the desired power in Takapoui et al. (2017). Depending on the horizon length T we choose only the first T elements of \bar{P}^{des} to sample P^{des} .

5.16 Results

We present the results in Table 4. Here there is a significant improvement in computation time compared to Gurobi. Also in this case the number of strategies $|\mathcal{S}|$ is much less than the number of variables or the worst case number of control input combinations. The maximum infeasibility and suboptimality are low even when the predictions are not exact and the performance of OCTs, OCT-Hs and NNs is comparable except in the last case.

6 Conclusions

We introduced the idea that using OCTs and OCT-Hs we obtain insight on the strategy of the optimal solution in many optimization problems as a function of key parameters. In this way, optimization is not a black box anymore, but rather it has a voice, i.e., we are able to provide insights on the logic behind the optimal solution. The class of optimization problems that these ideas apply to is rather broad since it includes any continuous and mixed-integer convex optimization problems without any assumption on the parameters dependency. The accuracy of our approach is in the 90–100%, while even when it does not provide the optimal solution the degree of suboptimality or infeasibility is extremely low. In addition, the computation times of our method are significantly faster than solving the problem using standard

Table 4 Hybrid control benchmarks

T	n_{var}	n_{con}	Learner	N	GT	$ S $	acc (%)	t_{ratio}	\bar{p}	\bar{d}
10	51	103	NN	10,000	3.00×10^{-4}	16	100.00	2.79	1.17×10^{-14}	3.78×10^{-15}
10	51	103	OCT	10,000	3.00×10^{-4}	16	99.00	0.12	9.37×10^{-4}	3.78×10^{-15}
10	51	103	OCT-H	10,000	3.00×10^{-4}	16	100.00	3.00	1.17×10^{-14}	3.78×10^{-15}
15	76	153	NN	10,000	4.00×10^{-4}	16	99.00	4.20	7.27×10^{-5}	4.29×10^{-15}
15	76	153	OCT	10,000	4.00×10^{-4}	16	99.00	3.68	7.27×10^{-5}	4.29×10^{-15}
15	76	153	OCT-H	10,000	4.00×10^{-4}	16	99.00	3.05	7.27×10^{-5}	4.29×10^{-15}
20	101	203	NN	10,000	6.00×10^{-4}	16	100.00	3.89	1.46×10^{-14}	6.04×10^{-15}
20	101	203	OCT	10,000	6.00×10^{-4}	16	99.00	4.16	2.23×10^{-4}	6.04×10^{-15}
20	101	203	OCT-H	10,000	6.00×10^{-4}	16	100.00	4.33	1.46×10^{-14}	6.04×10^{-15}
25	126	253	NN	10,000	2.00×10^{-4}	10	100.00	4.27	5.49×10^{-5}	6.33×10^{-15}
25	126	253	OCT	10,000	2.00×10^{-4}	10	100.00	4.38	5.49×10^{-5}	6.33×10^{-15}
25	126	253	OCT-H	10,000	2.00×10^{-4}	10	100.00	5.12	5.49×10^{-5}	6.33×10^{-15}
30	151	303	NN	10,000	1.00×10^{-4}	20	100.00	86.65	1.55×10^{-13}	7.72×10^{-16}
30	151	303	OCT	10,000	1.00×10^{-4}	20	100.00	148.37	1.55×10^{-13}	7.72×10^{-16}
30	151	303	OCT-H	10,000	1.00×10^{-4}	20	100.00	124.01	1.55×10^{-13}	7.72×10^{-16}
35	176	353	NN	10,000	3.00×10^{-4}	37	100.00	845.13	4.04×10^{-5}	1.14×10^{-15}
35	176	353	OCT	10,000	3.00×10^{-4}	37	98.00	1010.73	4.04×10^{-5}	3.48×10^{-4}
35	176	353	OCT-H	10,000	3.00×10^{-4}	37	100.00	1250.19	4.04×10^{-5}	5.98×10^{-5}
40	201	403	NN	10,000	2.00×10^{-4}	35	100.00	6461.46	6.50×10^{-14}	1.53×10^{-15}
40	201	403	OCT	10,000	2.00×10^{-4}	35	93.00	6846.49	6.50×10^{-14}	3.12×10^{-2}
40	201	403	OCT-H	10,000	2.00×10^{-4}	35	89.00	6551.66	6.50×10^{-14}	3.02×10^{-2}

optimization algorithms. Even though the solving times did not have a strict limit for the proposed applications, our method can be useful in online settings where problems must be solved within a limited time and with limited computational resources. Applications of this framework to fast online optimization appear in Bertsimas and Stellato (2019). Comparisons on several examples show that the out-of-sample strategy prediction accuracy of OCT-Hs is comparable to the NNs. OCTs show competitive accuracy in most comparisons apart from the cases with higher number of strategies where the prediction performance degrades. Note that this is not necessarily related to the size of the problem but rather to how the parameters affect the optimal solution and, in turn, the total number of strategies. OCTs also exhibit low infeasibility and low suboptimality even when the prediction is not correct. In addition, OCTs have higher interpretability than OCT-Hs and NNs because of their structure with parallel splits, therefore, being required in applications where models must be explainable. For these reasons, our framework provides a reliable and insightful understanding of optimal solutions in a broad class of optimization problems using different machine learning algorithms.

Appendix 1: Optimal classification trees

OCTs and OCT-Hs developed by Bertsimas and Dunn (2017, 2019) are a recently proposed generalization of classification and regression trees (CARTs) developed by Breiman et al. (1984) that construct decision trees that are near optimal with significantly higher prediction accuracy while retaining their interpretability. Bertsimas and Dunn (2019) have shown that given a NN (feedforward, convolutional and recurrent), we can construct an OCT-H that has the same in sample accuracy, that is OCT-Hs are at least as powerful as NNs. The constructions can sometimes generate OCT-Hs with high depth. However, they also report computational results that show that OCT-Hs and NNs have comparable performance in practice even for depths of OCT-Hs below 10.

Architecture

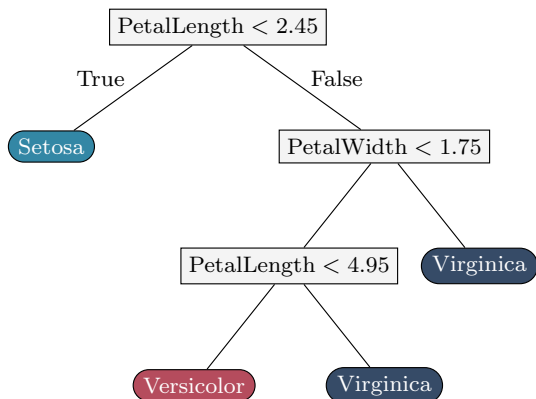
OCT recursively partition the feature space \mathbf{R}^p to construct hierarchical disjoint regions. A tree can be defined as a set of nodes $t \in \mathcal{T}$ of two types $\mathcal{T} = \mathcal{T}_B \cup \mathcal{T}_L$:

Branch nodes Nodes $t \in \mathcal{T}_B$ at the tree branches describe a split of the form $a_t^T \theta < b_t$ where $a_t \in \mathbf{R}^p$ and $b_t \in \mathbf{R}$. They partition the space in two subsets: the points on the left branch satisfying the inequality and the remaining ones points on the right branch. If splits involve a single variable we denote them as *parallel* and we refer to the tree as optimal classification tree (OCT).. This is achieved by enforcing all components of a_t to be all 0 except from one. Otherwise, if the components of a_t can be freely nonzero, we denote the splits as *hyperplanes* and we refer to the tree as optimal classification tree with-hyperplanes (OCT-H).

Leaf nodes Nodes $t \in \mathcal{T}_L$ at the tree leaves make a class prediction for each point falling into that node.

An example OCT for the Iris dataset appears in Fig. 8 (Bertsimas and Dunn 2019). For each new data point it is straightforward to follow which hyperplanes are satisfied and to make a prediction. This characteristic makes OCTs and OCT-Hs highly interpretable. Note that the level of interpretability of the resulting trees can be also tuned by changing

Fig. 8 Example OCT for the Iris dataset (Bertsimas and Dunn 2019)



minimum sparsity of a_t . The two extremes are maximum sparsity OCTs and minimum sparsity OCT-Hs but we can specify anything in between.

Learning

With the latest computational advances in MIO, Bertsimas and Dunn (2019) were able to exactly formulate the tree training as a MIO problem and solve it in a reasonable amount of time for problem sizes arising in real-world applications.

The OCT cost function is a tradeoff between the misclassification error at each leaf and the tree complexity

$$\mathcal{L}_{OCT} = \sum_{t \in \mathcal{T}_L} L_t + \alpha \sum_{t \in \mathcal{T}_B} \|a_t\|_1,$$

where the L_t is the misclassification error at node t and the second term represents the complexity of the tree measured as the sum of the norm of the hyperplane coefficients in all the splits. The parameter $\alpha > 0$ regulates the tradeoff. For more details about the cost function and the constraints of the problem, we refer the reader to Bertsimas and Dunn (2019, Sects. 2.2, 3.1).

Bertsimas and Dunn apply a local search method (Bertsimas and Dunn 2019, Sect. 2.3) that manages to solve OCT problems for realistic sizes in fraction of the time an off-the-shelf optimization solver would take. The algorithm proposed iteratively improves and refines the current tree until a local minimum is reached. By repeating this search from different random initialization points the authors compute several local minima and then take the best one as the resulting tree. This heuristic showed remarkable performance both in terms of computation time and quality of the resulting tree becoming the algorithm included in the OptimalTrees.jl Julia package (Bertsimas and Dunn 2019).

Appendix 2: Neural networks

NNs have recently become one of the most prominent machine learning techniques revolutionizing fields such as speech recognition (Hinton et al. 2012) and computer vision (Krizhevsky et al. 2012). The wide range of applications of these techniques recently extended to autonomous driving (Bojarski et al. 2016) and reinforcement learning (Silver et al. 2017). The popularity of neural networks is also due to the widely used open-source libraries learning on CPUs and GPUs coming from both academia and industry such as TensorFlow (Abadi et al. 2015), Caffe (Jia et al. 2014) and PyTorch (Paszke et al. 2017). We use feedforward neural networks which offer a good tradeoff between simplicity and accuracy without resorting to more complex architectures such as convolutional or recurrent neural networks (LeCun et al. 2015).

Architecture

Given L layers, a neural network is a composition of functions of the form

$$\hat{s} = f_L(f_{L-1}(\dots f_1(\theta))),$$

where each function consists of

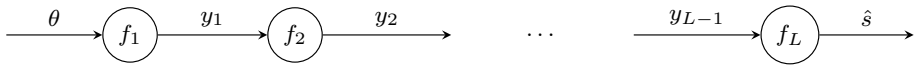


Fig. 9 Example feedforward neural network with functions f_i , $i = 1, \dots, L$ defined in (12)

$$y_l = f(y_{l-1}) = g(W_l y_{l-1} + b_l). \quad (12)$$

The number of nodes in each layer is n_l and corresponds to the dimension of the vector $y_l \in \mathbf{R}^{n_l}$. Layer $l = 1$ is defined as the *input* layer and $l = L$ as the *output* layer. Consequently $y_1 = \theta$ and $y_L = \hat{s}$. The linear transformation in each layer is composed of an affine transformation with parameters $W_l \in \mathbf{R}^{n_l \times n_{l-1}}$ and $b_l \in \mathbf{R}^{n_l}$. The activation function $g : \mathbf{R}^{n_l} \rightarrow \mathbf{R}^{n_l}$ models nonlinearities. We chose as activation function the rectified linear unit (ReLU) defined as

$$g(x) = \max(x, 0),$$

for all the layers $l = 1, \dots, L - 1$. Note that the max operator is intended element-wise. We chose a ReLU because on the one hand it provides sparsity to the model since it is 0 for the negative components of x and because on the other hand it does not suffer from the vanishing gradient issues of the standard sigmoid functions (Goodfellow et al. 2016). An example neural network can be found in Fig. 9.

For the output layer we would like the network to output not only the predicted class, but also a normalized ranking between them according to how likely they are to be the right one. This can be achieved with a softmax activation function in the output layer defined as

$$g(x)_j = \frac{e^{x_j}}{\sum_{j=1}^M e^{x_j}},$$

where $j = 1, \dots, M$ are the elements of $g(x)$. Hence, $0 \leq g(x) \leq 1$ and the predicted class is $\text{argmax}(\hat{s})$.

Learning

Before training the network, we rewrite the labels for the neural network learning using a one-hot encoding, i.e., $s_i \in \mathbf{R}^M$ where M is the total number of classes and all the elements of s_i are 0 except the one corresponding to the class which is 1.

We define a smooth cost function amenable to algorithms such as gradient descent, i.e., the cross-entropy loss

$$\mathcal{L}_{\text{NN}} = \sum_{i=1}^N -s_i^T \log(\hat{s}_i),$$

where log is intended element-wise. The cross-entropy loss \mathcal{L} can also be interpreted as the distance between the predicted probability density of the labels compared to the true one.

The actual training phase consists of applying stochastic gradient descent using the derivatives of the cost function using the back-propagation rule. This method works very well in practice and provides good out-of-sample performance with short training times.

References

- Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., et al. (2015). *TensorFlow: Large-scale machine learning on heterogeneous systems*. <https://www.tensorflow.org/>, software available from tensorflow.org.
- Alvarez, A. M., Louveaux, Q., & Wehenkel, L. (2017). A machine learning-based approximation of strong branching. *INFORMS Journal on Computing*, 29(1), 185–195.
- Bengio, Y. (2009). Learning deep architectures for AI. *Foundations and Trends® in Machine Learning*, 2(1), 1–127.
- Bengio, Y., Lodi, A., & Prouvost, A. (2018). *Machine learning for combinatorial optimization: A methodological tour d'horizon*. [arXiv:1811.0612](https://arxiv.org/abs/1811.0612).
- Bertsimas, D., & Dunn, J. (2017). Optimal classification trees. *Machine Learning*, 106(7), 1039–1082.
- Bertsimas, D., & Dunn, J. (2019). *Machine learning under a modern optimization lens*. London: Dynamic Ideas Press.
- Bertsimas, D., & Stellato, B. (2019). *Online mixed-integer optimization in milliseconds*. [arXiv:1907.02206](https://arxiv.org/abs/1907.02206).
- Bertsimas, D., & Tsitsiklis, J. N. (1997). *Introduction to linear optimization*. New York: Athena Scientific.
- Bojarski, M., Del Testa, D., Dworakowski, D., Firner, B., Flepp, B., Goyal, P., et al. (2016). *End to end learning for self-driving cars*. [arXiv:1604.17316](https://arxiv.org/abs/1604.17316).
- Bonami, P., Lodi, A., & Zarpellon, G. (2018). Learning a classification of mixed-integer quadratic programming problems. In W. J. van Hoeve (Ed.), *Integration of constraint programming, artificial intelligence, and operations research* (pp. 595–604). Cham: Springer.
- Boyd, S., & Vandenberghe, L. (2004). *Convex optimization*. Cambridge: Cambridge University Press.
- Breiman, L., Friedman, J. H., Olshen, R. A., & Stone, C. J. (1984). *Classification and regression trees. The Wadsworth statistics/probability series*. Monterey, CA: Wadsworth & Brooks/Cole Advanced Books & Software.
- Calafiore, G. C. (2010). Random convex programs. *SIAM Journal on Optimization*, 20(6), 3427–3464.
- Clarke, E., Gupta, A., Kukula, J., & Strichman, O. (2002). SAT based abstraction-refinement using ILP and machine learning techniques. In E. Brinksma & K. G. Larsen (Eds.), *Computer aided verification* (pp. 265–279). Berlin: Springer.
- Copeland, J. (2012). *Alan turing: The codebreaker who saved 'millions of lives'*. BBC News. <https://www.bbc.com/news/technology-18419691>. Online; posted 19-June-2012.
- Dai, H., Dai, B., & Song, L. (2016). Discriminative embeddings of latent variable models for structured data. In *Proceedings of the 33rd international conference on international conference on machine learning—volume 48, JMLR.org, ICML'16* (pp. 2702–2711).
- Dai, H., Khalil, E. B., Zhang, Y., Dilkina, B., & Song, L. (2017). *Learning combinatorial optimization algorithms over graphs*. [arXiv:1704.01665](https://arxiv.org/abs/1704.01665).
- Diamond, S., & Boyd, S. (2016). CVXPY: A Python-embedded modeling language for convex optimization. *Journal of Machine Learning Research*, 17(83), 1–5.
- Good, I. J. (1953). The population frequencies of species and the estimation of population parameters. *Biometrika*, 40(3/4), 237–264.
- Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. London: MIT Press.
- Gurobi Optimization Inc. (2020). *Gurobi optimizer reference manual*. <http://www.gurobi.com>.
- Hastie, T., Tibshirani, R., & Friedman, J. (2009). *The elements of statistical learning. No. 2 in Springer series in statistics*. New York: Springer.
- Hinton, G., Deng, L., Yu, D., Dahl, G. E., Mohamed, A., Jaitly, N., et al. (2012). Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6), 82–97.
- Hoffman, A. J. (1979). Binding constraints and helly numbers. *Annals of the New York Academy of Sciences*, 319(1 Second Intern), 284–288.
- Hutter, F., Hoos, H. H., Leyton-Brown, K., & Stützle, T. (2009). ParamLLS: An automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36(1), 267–306.
- Hutter, F., Hoos, H. H., & Leyton-Brown, K. (2011). Sequential model-based optimization for general algorithm configuration. In C. A. Coello (Ed.), *Learning and intelligent optimization* (pp. 507–523). Berlin: Springer.
- Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., & Darrell, T. (2014). *Caffe: Convolutional architecture for fast feature embedding*. [arXiv:1408.5093](https://arxiv.org/abs/1408.5093).
- Khalil, E. B., Bodic, P. L., Song, L., Nemhauser, G., & Dilkina, B. (2016). Learning to branch in mixed integer programming. In *Proceedings of the 30th AAAI conference on artificial intelligence, AAAI'16* (pp. 724–731). London: AAAI Press. <http://dl.acm.org/citation.cfm?id=3015812.3015920>.

- Klaučo, M., Kalúz, M., & Kvasnica, M. (2019). Machine learning-based warm starting of active set methods in embedded model predictive control. *Engineering Applications of Artificial Intelligence*, 77, 1–8.
- Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, & K. Q. Weinberger (Eds.), *Advances in neural information processing systems* (pp. 1097–1105). Curran Associates Inc.
- Kruber, M., Lübbecke, M. E., & Parmentier, A. (2017). Learning when to use a decomposition. In D. Salvagnin & M. Lombardi (Eds.), *Integration of AI and OR techniques in constraint programming* (pp. 202–210). Cham: Springer.
- LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature*, 521(7553), 436–444.
- López-Ibáñez, M., Dubois-Lacoste, J., Pérez Cáceres, L., Birattari, M., & Stützle, T. (2016). The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3, 43–58.
- Markowitz, H. (1952). Portfolio selection. *The Journal of Finance*, 7(1), 77–91.
- McAllester, D. A., & Schapire, R. E. (2000). On the convergence rate of Good–Turing estimators. In *Proceedings of the 13th annual conference on computational learning theory*.
- McDiarmid, C. (1989). *On the method of bounded differences*. London mathematical society lecture note series (pp. 148–188). Cambridge: Cambridge University Press.
- Minton, S. (1996). Automatically configuring constraint satisfaction programs: A case study. *Constraints*, 1(1–2), 7–43.
- Misra, S., Roald, L., & Ng, Y. (2019). *Learning for constrained optimization: Identifying optimal active constraint sets*. [arXiv:1802.09639v4](https://arxiv.org/abs/1802.09639v4).
- Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., et al. (2017). Automatic differentiation in PyTorch. In *NIPS-W*.
- Selsam, D., Lamm, M., Bünz, B., Liang, P., de Moura, L., & Dill, D. L. (2019). Learning a SAT solver from single-bit supervision. In *International conference on learning representations*.
- Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., et al. (2017). Mastering the game of go without human knowledge. *Nature*, 550(7676), 354–359.
- Smith, K. A. (1999). Neural networks for combinatorial optimization: A review of more than a decade of research. *INFORMS Journal on Computing*, 11(1), 15–34.
- Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction* (2nd ed.). New York: A Bradford Book.
- Takapoui, R., Moechle, N., Boyd, S., & Bemporad, A. (2017). A simple effective heuristic for embedded mixed-integer quadratic programming. *International Journal of Control*, 93, 1–11.
- Xu, L., Hutter, F., Hoos, H. H., & Leyton-Brown, K. (2008). SATzilla: Portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research*, 32(1), 565–606.
- Zheng, Y. S., & Federgruen, A. (1991). Finding optimal (s, S) policies is about as simple as evaluating a single policy. *Operations Research*, 39(4), 654–665. <https://doi.org/10.1287/opre.39.4.654>.