MIT Libraries | DSpace@MIT

MIT Open Access Articles

Global optimization: a machine learning approach

The MIT Faculty has made this article openly available. ***Please share*** how this access benefits you. Your story matters.

Citation: Bertsimas, D., Margaritis, G. Global optimization: a machine learning approach. J Glob Optim (2024).

As Published: https://doi.org/10.1007/s10898-024-01434-9

Publisher: Springer US

Persistent URL: https://hdl.handle.net/1721.1/157394

Version: Final published version: final published article, as it appeared in a journal, conference proceedings, or other formally published context

Terms of use: Creative Commons Attribution

Massachusetts Institute of Technology

Journal of Global Optimization
https://doi.org/10.1007/s10898-024-01434-9

Global optimization: a machine learning approach

Dimitris Bertsimas[1] · **Georgios Margaritis**[2]

Received: 23 August 2023 / Accepted: 30 August 2024
© The Author(s) 2024

Abstract
Many approaches for addressing global optimization problems typically rely on relaxations of nonlinear constraints over specific mathematical primitives. This is restricting in applications with constraints that are implicit or consist of more general primitives. Trying to address such limitations, Bertsimas and Ozturk (2023) proposed OCTHaGOn as a way of solving very general global optimization problems by approximating the nonlinear constraints using hyperplane-based decision-trees and then using those trees to construct a unified MIO approximation of the original problem. We provide extensions to this approach, by (i) approximating the original problem using other MIO-representable ML models besides decision trees, such as gradient boosted trees, multi layer perceptrons and suport vector machines (ii) proposing adaptive sampling procedures for more accurate ML-based constraint approximations, (iii) utilizing robust optimization to account for the uncertainty of the sample-dependent training of the ML models, (iv) leveraging a family of relaxations to address the infeasibilities of the final MIO approximation. We then test the enhanced framework in 81 global optimization instances. We show improvements in solution feasibility and optimality in the majority of instances. We also compare against BARON, showing improved optimality gaps and solution times in more than 9 instances.

Keywords Global optimization · Machine learning · Mixed integer optimization · Robust optimization

1 Introduction

Global optimizers aim to solve problems of the following form:

$$\begin{aligned} \min \quad & f(\boldsymbol{x}) \\ \text{s.t.} \quad & g_i(\boldsymbol{x}) \leq 0,\ i \in \bar{I}, \\ & h_j(\boldsymbol{x}) = 0,\ j \in \bar{J}, \\ & \boldsymbol{x} \in \mathbb{Z}^m \times \mathbb{R}^{n-m}, \end{aligned} \tag{1}$$

✉ Dimitris Bertsimas
dbertsim@mit.edu

Georgios Margaritis
geomar@mit.edu

[1] Sloan School of Management, Massachusetts Institute of Technology, Cambridge, MA 02139, USA

[2] Operations Research Center, Massachusetts Institute of Technology, Cambridge, MA 02139, USA

Published online: 07 October 2024

where f, g_i, h_i represent the objective function, the inequality constraints and the equality constraints respectively. The objective function and constraints may lack desirable mathematical properties like linearity or convexity, and the decision variables may be continuous or integer.

Most approaches in the global optimization literature, attempt to solve problem (1) by approximating it with more tractable optimization forms. For this purpose, they often use a combination of gradient-based methods, outer approximations, relaxations and mixed integer optimization (MIO). For instance, the popular nonlinear optimizer CONOPT uses a gradient-based approach in its solution process. As noted in [1], CONOPT finds an initial feasible solution using heuristics, performs a series of gradient descent iterations and then confirms optimality via bound projections. During the gradient-based part of the algorithm, CONOPT linearizes the constraints and performs a series of linear-search gradient-based iterations, while preserving feasibility at each step.

A different approach is the one detailed by [2], which uses outer approximations. This approach simplifies the problem by approximating the constraints via linear and nonlinear cuts, while preserving the initial feasible set of the problem. Such approach can only be used with constraints that obey a particular mathematical structure, such as linearity of integer variables and convexity of the nonlinear functions [3], or concavity and bilinearity [4], where the functions involved are amenable to efficient outer approximations. Although such approaches are effective in some scenarios, they haven't found extensive use as they they are restricted to certain classes of problems.

Another approach is the one used by the well-established commercial optimizer BARON, which combines MIO with convex relaxations and outer approximations. As detailed in [5], BARON uses a branch-and-reduce method, which partitions the domain of the constraints and objective into subdomains, and attempts to bound the decision variables in each subdomain depending on the mathematical form of the constraints. This approach produces a branch-and-bound tree that offers guarantees of global optimality, which is analogous to the branch-and-bound process used for solving MIO problems. A similar approach is used by the popular solver ANTIGONE. As described in [6], ANTIGONE first reformulates the problem, detects specific mathematical structure in the constraints and then uses convex underestimators in conjunction with a branch-and-cut method in order to recursively find the optimal solution.

Although all such approaches are very effective in dealing with certain types of global optimization problems, they each have their own weaknesses. For instance, gradient-based approaches depend on good initial feasible solutions and cannot easily handle integer variables. Also, approaches that leverage relaxations and outer approximations are restricted to specific types of nonlinearities, thus being ineffective against more general problems. Finally, approaches that use branching or MIO-based methods inevitably suffer from the curse of dimensionality due to their combinatorial nature.

At the same time, for reasons of efficiency, many global optimizers only allow constraints that use a subset of all the mathematical primitives. For instance, the optimizer BARON can only handle functions that involve $\exp(x)$, $\ln(x)$, x^a and b^x where a and b reals. However, real-world problems may contain constraints with a much richer set of primitives, such as trigonometric and piecewise-discontinuous functions, where it is not always possible to reduce the problem to a form compatible with the global optimizer. At the same time, such optimizers are generally ineffective in cases where the constraints lack an analytical representation.

Motivated by all those scenarios, [7] proposed OCTHaGOn, a global optimization framework that attempts to solve very general global optimization problems by combining Machine Learning with mixed integer optimization (MIO). The framework can be used in problems

that involve many types of constraints, such as convex, non-convex, and constraints that involve very general mathematical primitives. The main requirements of the method is that the decision variables involved in nonlinear constraints are bounded and that the nonlinear constraints can be evaluated quickly. Under these assumptions, OCTHaGOn first samples the nonlinear constraints for feasibility and trains a hyperplane-based decision tree (OCT-H, [8, 9]) on those samples. It does that for every non-linear constraint and then uses the MIO-representation of the resulting decision trees to construct an MIO approximation of the original problem. In then solves approximating MIO and uses a local-search projected gradient descent method to improve the solution in terms of both feasibility and optimality.

Despite its generality, the OCTHaGOn framework sometimes yields infeasible or suboptimal solutions due to being approximate in nature. Hence, in this paper, we provide extensions to OCTHaGOn in order to improve both feasibility and optimality of the generated solutions. We enhance the framework by leveraging adaptive sampling, robust optimization, constraint relaxations and a variety of MI-representable ML models to create better approximations of the original problem. We test the enhanced framework in a number of Global Optimization benchmarks, and we show improved optimality gaps and solution times in the majority of test instances. We also compare against the commercial optimizer BARON, showing improvements in a number of instances.

1.1 Machine learning in optimization

The interplay between machine learning (ML) and mathematical optimization has been an active area recently. [10] provides a survey of different optimization methods from an ML standpoint. In another survey ([11]), the authors show how fundamental ML tasks, such as classification, regression and clustering can be formulated as optimization problems. Additionally, a lot of emphasis has been placed on using ML for aiding the formulation and solution of optimization problems. For instance, ML has been used to assist with the solution of difficult and large-scale optimization formulations (e.g. [12, 13]), and it has also been used to speed-up tree search search [14] and guide branching in MIP problems [15].

Apart from the general interplay of ML and optimization, an area that has garnered a lot of interest is the incorporation of ML models as surrogates into an optimization problem. Namely, [16] explores various formulations for representing linear model decision trees in an optimization context, [17] provides efficient formulations for the MIO-representation of tree ensembles, and [18] provides strong formulations for representing ReLU neural networks. Open-source frameworks have also been proposed for representing ML models using MIO formulations, including optimization with constraint learning (OptiCL, [19]) and optimization and machine learning toolkit (OMLT, [20]).

The idea of using models as optimization surrogates has also been applied to some extent in the context of global optimization. The solver ARGONAUT ([21]) uses constrained sampling techniques to build surrogate representations of unknown constraints, which are then globally optimized. Although this approach bears some resemblance to ours, we employ very different sampling and solution techniques, we use standard ML surrogate models while ARGONAUT uses analytical surrogate functions (e.g. quadratic, signomial), and we also end up with a linear MIO representation, while ARGONAUT constructs a nonlinear MIO approximation. Another approach with similar differences to ours is the ALAMO framework ([22]), which attempts to learn algebraic functions from data by solving nonlinear MIO problems.

Another framework that uses data-driven surrogates to solve global optimization problems is OCTHaGOn [7]. OCTHaGOn samples the non-linear constraints and learns the non-

linearities using MIO representable decision trees with hyperplanes (OCT-Hs/ORTs, [8, 9]). The trained decision trees are embedded as constraints into a mixed integer optimization (MIO) formulation that approximates the original problem. After a solution to the MIO is obtained, a local repair is performed to address for infeasibility and suboptimality which stem from the approximation errors of the decision trees. In our work, we extend this method and provide significant improvements as outlined in Sect. 1.2.

1.2 Contributions

Our approach for solving global optimization problems is an improvement of OCTHaGOn [7] with the following key differences and enhancements:

(i) **Different ML models:** OCTHaGOn only uses hyperplane-based decision trees (OCT-H) as constraint approximators, but we extend the approach to other types of MIO-representable ML models, such as support vector machines, gradient boosted trees, and multi-layer perceptrons. We also employ a method for selecting which of those models to use for which constraint.
(ii) **Adaptive sampling:** We propose an adaptive sampling procedure that helps generate high-quality samples of the nonlinear constraints for more accurate constraint approximations. We then demonstrate that this method is very effective in reducing feasibility gaps of the obtained solutions.
(iii) **MIO relaxations:** The ML-based MIO approximation of the original problem can sometimes be infeasible. To eliminate such infeasibilities, we relax constraints of the resulting MIO in a way that makes the MIO approximation feasible if the original problem is also feasible.
(iv) **Robust optimization:** We utilize robust optimization to account for the uncertainty of the sample-dependent training of the ML models. In particular, we attempt to model the uncertainty in the trained parameters of the ML approximators, and we then use robust optimization to correct for this uncertainty.

We test the framework on 81 global optimization instances, with 77 of those being part the standard benchmarking library MINLPLib. We show that our enhancements reduce the optimality gaps of OCTHaGOn in 36 instances, while increasing the gap in only 5. We also show that in 9 out of the 81 instances, the enhanced framework provides better or faster solutions than BARON.

Overall, we demonstrate that despite its heuristic nature, the enhanced framework, is a promising method for finding globally optimal solutions in a variety of instances. The framework can be applied in problems with very general convex and non-convex constraints, which may include a wide range of mathematical primitives Hence, due to its generality, the method can be used in problems that are incompatible with traditional global optimizers such as BARON and ANTIGONE.

2 OCTHaGOn

Since our method is an extension of OCTHaGOn, we will use this section to present the steps involved in the OCTHaGOn framework. OCTHaGOn [7] attempts to solve Global Optimization problems using a combination of ML and MIO. It relies on creating decision-tree based approximations of the nonlinear constraints, embedding those approximations in

an approximating MIO, and then solving and improving the solution resulting from that MIO to obtain a high-quality solution of the original problem.

Due to its nature, OCTHaGOn can also be used to solve problems with very general constraints. The only requirement is that the user of the framework supplies bounds for the variables involved in the nonlinear constraints. If such bounds are not provided, the quality of the solution returned by the framework can be adversely affected.

2.1 Algorithmic process

OCTHaGOn addresses problems of the following form:

$$\begin{aligned} \min \quad & f(\boldsymbol{x}) \\ \text{s.t.} \quad & g_i(\boldsymbol{x}) \leq 0,\ i \in \bar{I}, \\ & h_j(\boldsymbol{x}) = 0,\ j \in \bar{J}, \\ & \boldsymbol{x} \in \mathbb{Z}^m \times \mathbb{R}^{n-m}, \end{aligned} \tag{2}$$

where the functions f, g_i, h_j can be non-convex.

OCTHaGOn attempts to create and solve an MIO approximation of Problem (2) in the following steps:

1. **Standard form problem generation:** In order to create the MIO approximation of problem (2), we first separate the linear from the non-linear constraints. We also find the constraints that define explicit bounds $[\underline{x}_k, \overline{x}_k]$ for every optimization variable x_k that is involved in a nonlinear constraint. If a variable x_k doesn't have an explicit bound (either from above or below), we attempt to compute that bound by minimizing/maximizing that variable subject to the linear constraints of (2). Note that having explicit variable bounds is essential for the step 2 of the methodology. Then, after determining the variable bounds, we rewrite our original problem into the following form:

$$\begin{aligned} \min_{\boldsymbol{x}} \quad & f(\boldsymbol{x}) \\ \text{s.t.} \quad & g_i(\boldsymbol{x}) \leq 0,\ i \in I, \\ & h_j(\boldsymbol{x}) = 0,\ j \in J, \\ & \boldsymbol{A}\boldsymbol{x} \geq \boldsymbol{b},\ \boldsymbol{C}\boldsymbol{x} = \boldsymbol{d}, \\ & x_k \in [\underline{x}_k, \overline{x}_k],\ k \in [n]. \end{aligned} \tag{3}$$

   The linear constraints of this resulting formulation are directly passed to the MIO approximation, while the non-linear ones are approximated using steps 2–6.
2. **Sample and evaluate nonlinear constraints:** For each nonlinear constraint $g_i(\boldsymbol{x}) \leq 0$, we generate samples of the form $D_i = \{(\widetilde{\boldsymbol{x}}_{\boldsymbol{k}}, \tilde{y}_k)\}_{k=1}^n$ where $\tilde{y}_k = \mathbb{1}\{g(\widetilde{\boldsymbol{x}}_{\boldsymbol{k}}) \leq 0\}$ and $\mathbb{1}\{\cdot\}$ is the indicator function. In order to chooses the points $\widetilde{\boldsymbol{x}}_{\boldsymbol{1}}, \ldots, \widetilde{\boldsymbol{x}}_{\boldsymbol{k}}$ where the samples are evaluated, the framework uses the following sampling methods: boundary sampling, latin hypercube sampling (LH sampling) and kNN Quasi-Newton sampling. Boundary sampling samples the corners of the hyper-rectangle that is formed by the bounds of the decision variables (i.e. for each decision variable x_j of the constraint, we have that $x_j \in [\underline{x}_j, \bar{x}_j]$ after computing bounds in step 1). LH sampling, due to its space-filling characteristics, is then used to obtain constraint samples from the interior of the decision-variable hype-rectangle (i.e. each decision variable x_j is sampled in the range $[\underline{x}_j, \bar{x}_j]$). Finally, kNN Quasi-Newton sampling is proposed by the authors as a way to sample

the boundaries of the constraints, given the already generated samples from the previous steps.

3. **Train decision tree:** Hyperplane based decision trees (OCT-H, [8, 9]) are trained on the datasets D_i to approximate the feasibility space of the nonlinear constraint $g_i(\boldsymbol{x}) \leq 0$. This yields decision tree aproximators $\bar{c}_i(\boldsymbol{x})$ such that $\bar{c}_i(\boldsymbol{x}) \simeq \mathbb{1}\{g_i(\boldsymbol{x}) \leq 0\}$. If we have a non-linear objective, then this objective is approximated with regression trees (ORT-H, [8, 9]) instead of classification trees.
4. **Generate mixed-integer (MIO) approximation:** The trained decision trees $\bar{c}_i(\boldsymbol{x})$ are transformed into an efficient MIO representation by extracting the corresponding hyperplane splits and then using disjunction formulations. The resulting representations are then embedded as constraints of the form $\bar{c}_i(\boldsymbol{x}) = 1$ into the approximation MIO of problem (2).
5. **Solve MIO approximation:** The MIO approximation of problem (2) is solved using commercial solvers, such as IBM CPLEX and Gurobi.
6. **Check and improve solution:** The MIO is just an approximation of the original global optimization problem, so the solution of step 5 can be near-optimal and near-feasible. In order to account for this, the framework first measures the feasibility of the solution with respect to the nonlinear constraints, then it computes the gradient of the nonlinear objective and constraints (i.e. using automatic differentiation), and finally it performs a sequence of projected gradient descent steps to help improve feasibility and optimality of the solution.

The authors of OCTHaGOn test the framework against various benchmarks of the MINLP library, and demonstrate that in many cases, it works well compared to traditional global optimizers such as CONOPT, IPOPT and BARON.

3 Enhancements

In this section, we propose a variety of improvements on top of OCTHaGOn which improve the quality of the solutions generated by the framework. We first provide a brief outline of the solutions steps of the enhanced framework. For each step, we also indicate whether the step is the same as OCTHaGOn, whether it is an enhanced version of an OCTHaGOn step, or whether it is a new step. The outline of the different steps are shown below:

1. **Standard form problem generation:** We separate the linear from the nonlinear constraints and attempt to compute bounds for the variables involved in the nonlinear constraints. Then, we initialize the MIO approximation of the original problem with the linear and bound constraints. The non-linear constraints will be approximated and included in this MIO approximation in steps 2–4.
2. **Sample and evaluate nonlinear constraints (enhanced step):** We sample the nonlinear constraints in the following 4 steps: (i) Boundary sampling, (ii) latin hypercube sampling, (iii) KNN Quasi-Newton sampling and (iv) OCT-based adaptive sampling. The first 3 sampling steps are the same as the ones used in OCTHaGOn and are used to obtain an initial set of samples. Then, we use OCT-based adaptive sampling, which is part of our enhancements, to adaptively obtain high-quality samples in areas where the nonlinear constraints are not approximated well by the ML learners (e.g. near the constraint boundaries).
3. **ML model training (enhanced step):** For each nonlinear constraint, we train a series of MIO-representable ML models (namely OCT-Hs, MLPs, SVMs and GBMs) on the

samples generated during step 2. We measure the accuracy of those models on a test set, and for each nonlinear constraint, we pick the model that demonstrates the best predictive performance. We then use the model for approximating the corresponding nonlinear constraint. We follow a similar process to approximate the non-linear objective (if such objective is present), but in this case, we use regression instead of classification models. The enhancement in this step is the training of multiple ML models (MLPs, SVMs, GBMs), besides the decision trees (OCT-Hs) which are the only models used by OCTHaGOn.

4. **Generate MIO approximation (enhanced step):** For each one of the nonlinear constraints, we take the corresponding ML approximator from the last step and we represent it using an MIO formulation. Then, we embed all the MIO approximations into a unified MI formulation. In this formulation, we also include the linear constraints (and objectives) of the original problem. The enhancement in this step is that besides the decision trees (OCT-Hs) used by OCTHaGOn, we also formulate and represent the additional ML models (MLPs, SVMs and GBMs) using a MIO formulation.
5. **Introduce robustness (new step):** In order to account for the uncertainty in the trained parameters of the ML constraint approximators, we utilize Robust Optimization. We assume that the trained parameters of SVMs, GBMs and OCT-Hs lie in p-norm uncertainty sets, and we modify the MIO representation of the learners by considering the corresponding robust counterparts. The level of robustness is controlled by the parameter ρ, where a value of $\rho = 0$ means that we don't account for robustness, while higher values of ρ lead to more robust and more conservative solutions.
6. **Constraint relaxation (new step):** In certain cases, the MIO approximation of the original problem is infeasible, due to the inexact nature of the constraint approximators. To account for this issue, we relax the MIO constraints, while introducing a relaxation penalty into the objective.
7. **Solve MIO approximation:** We solve the MIO approximation of the original problem using commercial solvers, such as IBM CPLEX and Gurobi.
8. **Improve solution (enhanced step):** Since the MIO solution can be near-optimal or near-feasible, we perform a series of projected gradient descent iterations in order to help improve feasibility and optimality. In this PGD step, we use the same procedure as in OCTHaGOn, but we also use momentum in order to reduce the chances of landing in local optima.

In the next parts of Sect. 3, we will analyze and justify each of the proposed enhancements we made.

3.1 Enhancement 1: ML model training

One of the key aspects of OCTHaGOn framework is the approximation of nonlinear constraints using MIO-representable ML models. For this, OCTHaGOn only uses hyperplane-based decision trees (OCT-H and ORT-H, [8, 9]) which are improved generalizations of classification and regression trees (CARTs). However, there are many more MIO-representable ML models. In this work, besides decision trees, we also utilize support vectors machines (SVM), gradient boosting machines (GBMs) and multi-layer perceptrons with ReLU activations (MLPs), all of which are representable using a linear MIO formulation. The way we train and embed the different models into an MIO formulation is described below:

- **Support vector machines:** Support vector machines are ML models that use a suitable hyperplane to make predictions, either for classification [23] or regression [24]. For the purpose of this work, we are only considering the case of linear SVMs and we are not using any type of kernel, since our goal is to generate linear and tractable MIO approximations of the constraints. In fact, after training a linear SVM, we can easily embed its predictions into a linear MIO by adding the following linear constraint:

$$y_{SVM} = \beta_0 + \boldsymbol{x}^T \boldsymbol{\beta}, \tag{4}$$

  where $\boldsymbol{\beta}$, β_0 are the trained model parameters of the SVM.
  More information about training and embedding the SVM models can be found in "Appendix A.1".
- **Decision trees:** Decision trees (DTs) are ML models that partition the observation space into disjoint leaves after a sequence of splits. Due to their inherent interpretability and their ability to model non-linear relationships in data, they have been used a lot in practice. [25] first introduced classification and regression trees (CART), a framework that greedily partitions the observation space using parallel splits (i.e. splitting by 1 feature at a time). Ever since, decision tree models have been revised and extended.
  In this work, we use a generalized version of DTs which also allow for hyperplane splits instead of just parallel splits (OCT-H, [8]). In this setting, each intermediate node of the tree is associated with a split of the form $\boldsymbol{a}_i^T \boldsymbol{x} \leq b_i$ with $\boldsymbol{a}_i \in \mathbb{R}^n$. Training of the the hyperplane-based DTs is done the Interpretable AI software [26]. Then, following [7], we can represent the output of the generalized DT with the following big-M formulation:

$$\begin{aligned}
&\sum_{i=1}^{|\mathcal{L}|} z_i p_i = y_{DT}, \\
&\sum_{i=1}^{|\mathcal{L}|} z_i = 1, \\
&\boldsymbol{a}_j^T \boldsymbol{x} \leq b_j + M(1 - z_i), \quad \forall i \in \mathcal{L}, j \in L(L_i), \\
&\boldsymbol{a}_j^T \boldsymbol{x} \geq b_j - M(1 - z_i) + \epsilon, \quad \forall i \in \mathcal{L}, j \in R(L_i).
\end{aligned} \tag{5}$$

  Here, $\mathcal{L}$ is the set of leaves, p_i is the prediction at leaf i, $L(L_i)$ is the set of intermediate nodes that have leaf i on their left subtree and $R(L_i)$ is the set of intermediate nodes that have leaf i on their right subtree.
  More Information about representing and embedding DTs using an MIO formulation can be found in "Appendix A.2".
- **Gradient boosting machines:** Gradient boosted machines (GBM) can be represented as ensembles of base-learners, where each learner is trained sequentially to correct the mistakes of the previous learner. After training, if we use y_i to denote the prediction of the i-th learner, then the output of the GBM model can be represented as:

$$y = \sum_{i=1}^{n} a_i y_i, \tag{6}$$

  where a_i are weights associated with each learner. In theory, many different types of models can be used as based-learners for GBM, but we will limit our focus to decision trees, which are used most often in practice. There are multiple ways we can represent a trained GBM ensemble using an MIO formulation. The simplest way is to embed each

tree independently using Formulation (5) and then link the tree predictions using Eq. (6). This method is used in [19] and has the problem that it uses big-Ms and ϵ constants, which may lead to weaker formulations with numerical instabilities.
For this reason, we instead use the ensemble formulation presented in [17], which avoids these problems. Then, to connect this formulation with the decision variables $\boldsymbol{x}$ of the original problem, we use the linking constraints presented in [27]. The detailed formulation can be found in "Appendix A.3".

- **Neural networks:** Neural networks have been shown to be very powerful models both on classification and regression tasks. Although not all types of Neural Networks are MIO representable, some of them are. For the purpose of this work, we will limit our focus to multi-layer perceptrons (MLP) with rectified linear unit activations (ReLU). This class of models are MIO-representable through a linear big-M formulation [28].
In our work, we use a standard big-M formulation shown in [19]. The exact formulation can be found in "Appendix (A.4)". Better formulations for representing MLPs have also been proposed. Most notably, the authors on [18] propose an ideal MIO formulation without continuous auxiliary variables. However, we opt for the simpler big-M formulation since its significantly easier to embed into our framework without additional solution steps. At the same time, in the types of global-optimization instances we test our framework, the bottleneck is sampling and model training and not MIO solution time. Thus, the choice of formulation does not have a significant effect in the overall performance of the framework.

Based on the above, we can use multiple MIO-representable models to approximate a nonlinear objective or constraint. Choosing which model type to use for each nonlinear constraint (or objective) is an important task, since this choice heavily determines the quality of the resulting MIO approximation. Hence, following [19], we employ a cross-validation procedure to determine the best model type for each nonlinear function. For each nonlinear objective/constraint, we train a model of each type, and we use R^2/accuracy to select the best model that will be used for the approximation. Then, we construct a MIO approximation of the following form:

$$\begin{aligned} \min_{\boldsymbol{x}\in\mathbb{R}^n} \quad & y_{ML\_REGR}(\boldsymbol{x}) \\ \text{s.t.} \quad & y^{(i)}_{ML}(\boldsymbol{x}) \geq a_i, \ i \in I, \\ & \boldsymbol{C}\boldsymbol{x} \leq \boldsymbol{d}. \end{aligned} \tag{7}$$

Here $y_{ML\_REGR}(\boldsymbol{x})$ is the MIO-representation of the regressor that best approximates the objective. Then, $y^{(i)}_{ML}(\boldsymbol{x})$ is the MIO-representation of the classifier that best approximates the constraint $g_i(\boldsymbol{x}) \leq 0$ and a_i is a fixed threshold that depends on the classifier model type. For instance, for MLPs we have that $y^{(i)}_{ML}(\boldsymbol{x}) = y_{MLP}$ and $a_i = 0$, and for GBMs we have $y^{(i)}_{ML}(\boldsymbol{x}) = y_{GBM}$ and $a_i = 0.5$.

3.2 Enhancement 2: sampling

In order to have good approximations of the nonlinear constraints, we need high quality samples. The samples need to capture the non-linear and non-convex feasible regions of the constraints, which means that a static sampling procedure (such as uniform sampling) is not enough. In particular, the sample-generation process should not be agnostic to the sampled constraint, but it should instead adapt to the landscape of the function we are attempting to sample.

The sampling procedure of OCTHaGOn involves three steps: Boundary Sampling, LH sampling and kNN Quasi-Newton sampling. Both Boundary Sampling and LH sampling are static sampling procedures, since they sample all functions in the same manner, without taking into account the values of those functions to determine where to sample. On the other hand, kNN Quasi-Newton sampling is an adaptive sampling procedure which attempts to sample the constraints on the feasibility boundary.

For our approach, we initially use the same three steps as OCTHaGOn in order to generate a first set of samples. Then, we propose an adaptive sampling method in order to generate even more fine-grained samples for the constraints. Our method is called *OCT sampling* and attempts to iteratively resample the constraint in areas that the learners have high uncertainty in approximating. Those can be areas where we don't have many samples but we have nonlinearities and mixed-feasibility regions, making it harder for the learners to approximate. To detect such areas, we train OCT-H learners in different subsets of samples, and we find regions where there is a high prediction disagreement between the trained learners.
A brief outline of the method is decribed below (detailed algorithm in "Appendix A.5)":

1. We start with an initial set of samples generated using Boundary Sampling, LH sampling and kNN Quasi-Newton sampling.
2. We train different OCT-H trees on different subsets/perturbations of the current samples.
3. We find polyhedral areas where there is high prediction disagreement between the trees. These areas are intersections of tree leaves.
4. We resample those polyhedral areas using hit-and-run sampling [29].
5. We repeat the process from Step 2 to obtain even more samples.

An example of implementing OCT Sampling is shown in Fig. 1. By applying Steps 1–3 in this example, our method first finds the region where the OCT-H trees have high prediction disagreement. This region is a union of polyhedra and is shown in red in Fig. 1a. Then, the method samples the region using hit-and-run sampling as shown in Fig. 1b.

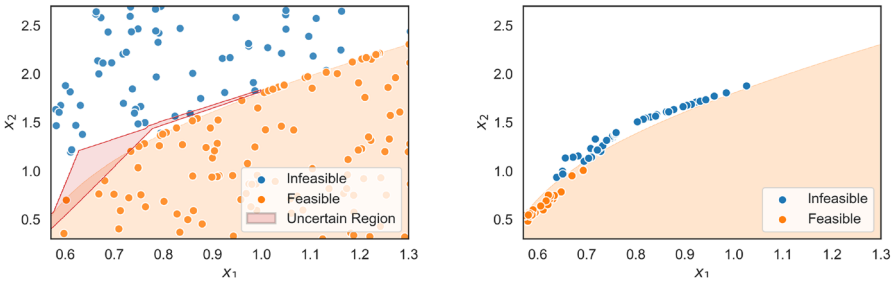
We notice that in this example, the uncertain region has a number of properties. First, there are very few samples inside the region. Secondly, the region is located on a mixed-feasibility area of the constraint (i.e. very close to the decision boundary). Thirdly, the region captures the most nonlinear part of the constraint. All of those properties make the region ideal for sampling, since the lack of samples, the mixed feasibility and the nonlinearity in this area means that region is hard to approximate by the learners.

We also notice that in this example, there are multiple regions that satisfy one of those properties, but very few that satisfy all three. For instance, we can easily see regions that are not well sampled, but those regions are not chosen since they are on the interior of the constraint (and thus easier to predict correctly by all the OCT models). There are also other regions on the feasibility boundary which are not chosen since they are well sampled (and thus the OCT models agree on their predictions in those regions).

3.3 Enhancement 3: relaxations

As we described in previous sections, in order to solve global Optimization Problems, we first create a MIO approximation of the original problem and we then optimize over this MIO in order to receive an approximate solution $\boldsymbol{x}_*$. Then, we perform a local-search PGD step in order to repair the solution $\boldsymbol{x}_*$ in terms of both feasibility and optimality.

However, due to the nature of this approach, it may occur that the MIO approximation is not feasible, even if the original problem is feasible. For instance, if a constraint is very difficult to satisfy, then we may have very limited feasible samples of that constraint. Thus



(a) Initial samples & uncertain region. (b) Samples resulting from OCT Sampling.

Fig. 1 Behavior of OCT sampling on a nonlinear constraint. OCT Sampling first detects the regions of high prediction uncertainty (shown in red) and then samples the region using hit-and-run sampling. (Color figure online)

the learner may approximate the constraint as infeasible almost everywhere, leading to an infeasible MIO.

To mitigate this issue, we relax the approximating MIO so that it will always have a feasible solution, provided that the original problem is also feasible. In particular, given the approximation MIO of Eq. (7), we relax it as follows:

$$\begin{aligned} \min_{\boldsymbol{x}\in\mathbb{R}^n, u\in\mathbb{R}} \quad & y_{ML\_REGR}(\boldsymbol{x}) + \lambda \sum_{i\in I} u_i \\ \text{s.t.} \quad & y^{(i)}_{ML}(\boldsymbol{x}) + u_i \geq a_i, \; i \in I, \\ & \boldsymbol{C}\boldsymbol{x} \leq \boldsymbol{d}, \\ & u_i \geq 0, \quad i \in I, \end{aligned} \tag{8}$$

where u_i are the relaxation variables and λ is a parameter that determines how much to penalize relaxed constraints. By including this relaxation variable in the constraints, we are ensuring that the MIO approximate constraints won't introduce any infeasibilities. Also, the more we increase the penalty λ, the more the MIO solver will prioritize solutions with small values of the variable u_i and thus less relaxed constraints will be. However, we also have to note that we first attempt to solve the problem without relaxations, and if the approximating MIO is infeasible, then we resolve using the described relaxation technique.

3.4 Enhancement 4: robustness

When using ML models to approximate nonlinear functions through samples, there can be a great deal of uncertainty in the learned model parameters. In particular, training on a different set of samples can lead to different model parameters and thus different approximations of the nonlinear function. To partially account for this issue, we will use Robust Optimization when embedding the ML learners into the final MIO.

Robust optimization (RO) is a methodology for dealing with uncertainty in the data of an optimization problem [30]. In our approach, however, we will use RO to deal with uncertainty not in the model data, but in the trained model parameters. For all types of models, we chose to model uncertainty using the p-norm uncertainty set $\mathcal{U}_p^\rho = \{\boldsymbol{x} \in \mathbb{R}^n : ||\boldsymbol{x}||_p \leq \rho\}$. The precise way of defining the uncertainty is slightly different across the learners, so we will describe each of the different models separately:

- **Support vector machines:** In the case of SVMs, we attempt to model the fact that after training the SVM, there is uncertainty in the values of the coefficient vector $\boldsymbol{\beta}$. For our use-case, we will choose to model $\boldsymbol{\beta}$ using multiplicative uncertainty, in order to scale the coefficients of $\boldsymbol{\beta}$ proportionally to their nominal values. More concretely, if an SVM is used to approximate a constraint $g(\boldsymbol{x}) \leq 0$ and $\bar{\beta}_0$ and $\bar{\boldsymbol{\beta}}$ are the nominal parameters of the SVM after training, then we approximate $g(\boldsymbol{x}) \leq 0$ with the uncertain constraint:

$$\bar{\beta}_0 + (\bar{\boldsymbol{\beta}} \odot (\mathbf{1} + z))^T \boldsymbol{x} \geq 0, \quad \forall z \in \mathcal{U}_p^\rho, \tag{9}$$

  where $\odot$ denotes the element-wise vector product and $\mathbf{1}$ is the vector of ones. Then, by using the tools presented in [31], this constraint can be written equivalently as:

$$\bar{\beta}_0 + \bar{\boldsymbol{\beta}}^T \boldsymbol{x} - \rho ||\bar{\boldsymbol{\beta}} \odot \boldsymbol{x}||_q \geq 0, \tag{10}$$

  where $||\cdot||_q$ is the dual norm of $||\cdot||_p$ (i.e. $\frac{1}{q} + \frac{1}{p} = 1$).
- **Decision trees:** gradient boosted machines In the case of decision trees, we will model the uncertainty in the coefficient vectors $\boldsymbol{a}_j$ of the hyperplane splits. As with SVMs, we will again use multiplicative uncertainty to scale the coefficients proportionately to their nominal values. In this case, multiplicative uncertainty is crucial, as it allows us to keep the zero elements of the coefficient vectors constant. For instance, if the original decision tree uses parallel splits, we only want to consider uncertain vectors $\boldsymbol{a}_j$ that represent parallel splits, an effect which will be captured by using multiplicative uncertainty.
  More formally, if we want to approximate the constraint $g(\boldsymbol{x}) \leq 0$ with a decision tree and $\bar{\boldsymbol{a}}_j$ is the coefficient vector of the j node of the tree after training, then the splitting constraints of Eq. (5) will take the following form after accounting for uncertainty:

$$\begin{aligned} &(\bar{\boldsymbol{a}}_j \odot (\mathbf{1} + \boldsymbol{u}))^T \boldsymbol{x} \leq b_j + M(1 - z_i), \quad \forall \boldsymbol{u} \in \mathcal{U}_p^\rho, \quad \forall i \in \mathcal{L}, j \in L(L_i), \\ &(\bar{\boldsymbol{a}}_j \odot (\mathbf{1} + \boldsymbol{u}))^T \boldsymbol{x} \geq b_j - M(1 - z_i) + \epsilon, \quad \forall \boldsymbol{u} \in \mathcal{U}_p^\rho, \quad \forall i \in \mathcal{L}, j \in R(L_i), \end{aligned} \tag{11}$$

  Again, by using the tools in [31], we can write the above constraints equivalently as:

$$\begin{aligned} &\bar{\boldsymbol{a}}_j^T \boldsymbol{x} + \rho ||\bar{\boldsymbol{a}}_j \odot \boldsymbol{x}||_q \leq b_j + M(1 - z_i), \quad \forall i \in \mathcal{L}, j \in L(L_i), \\ &\bar{\boldsymbol{a}}_j^T \boldsymbol{x} - \rho ||\bar{\boldsymbol{a}}_j \odot \boldsymbol{x}||_q \geq b_j - M(1 - z_i) + \epsilon, \quad \forall i \in \mathcal{L}, j \in R(L_i), \end{aligned} \tag{12}$$

  where $||\cdot||_q$ is the dual norm of $||\cdot||_p$.

4 An illustrative example

To better illustrate the different steps of the enhanced optimization methodology, we consider the following non-convex problem:

$$\begin{aligned} \min_{\boldsymbol{x} \in \mathbb{R}^n} \quad & f(\boldsymbol{x}) = -x_2 \\ \text{s.t.} \quad & g_1(\boldsymbol{x}) = -0.43 \ln(x_1 - 0.5) - 1.1 - x_1 + x_2 \leq 0, \\ & g_2(\boldsymbol{x}) = -x_2 + 0.33 \ln(x_1 - 0.4) + 1.2 - 0.2 x_1 \leq 0, \\ & g_3(\boldsymbol{x}) = -x_2 + 1.1 x_1 + 0.3 \leq 0, \\ & g_4(\boldsymbol{x}) = -x_2 - 1.5 x_1 + 2.6 \leq 0, \\ & 0.51 \leq x_1 \leq 1.5, \\ & 0.3 \leq x_2 \leq 1.6. \end{aligned} \tag{13}$$

To facilitate understanding of the method, we chose a problem with 2 variables and a linear objective, although the methodology also supports non-linear and non-convex objectives.

4.1 Standard form generation

The first step is to separate the linear from the non-linear constraints and bring the problem into the standard form described in Eq. (3). After doing that, Problem (13) becomes:

$\min f(\boldsymbol{x}) = -x_2$	Objective
s.t. $g_1(\boldsymbol{x}) = -0.43\ln(x_1 - 0.5) - 1.1 - x_1 + x_2 \leq 0,$	Nonlinear
$g_2(\boldsymbol{x}) = -x_2 + 0.33\ln(x_1 - 0.4) + 1.2 - 0.2x_1 \leq 0,$	constraints
$g_3(\boldsymbol{x}) = -x_2 + 1.1x_1 + 0.3 \leq 0,$	Linear
$g_4(\boldsymbol{x}) = -x_2 - 1.5x_1 + 2.6 \leq 0,$	constraints
$0.51 \leq x_1 \leq 1.5,$	Variables
$0.3 \leq x_2 \leq 1.6.$	and bounds

The linear constraints and the linear objective are directly passed to the MIO model. The nonlinear constraints involving g_1 and g_2 will be approximated in the next steps of the methodology. However, before proceeding into the next steps, we make sure that all variables of the nonlinear constraints are bounded. If they aren't, then we attempt to compute bounds the way that is described in Sect. 2

4.2 Sampling of nonlinear constraints

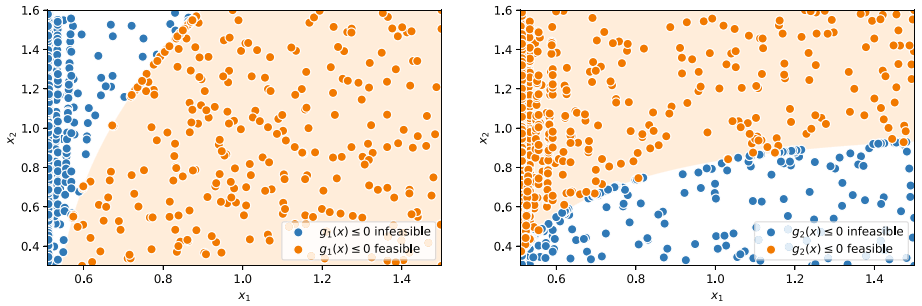
In this step, we sample the nonlinear constraints $g_1(\boldsymbol{x}) \leq 0$ and $g_2(\boldsymbol{x}) \leq 0$. The goal is to obtain samples $\{(\tilde{\boldsymbol{x}}_{\boldsymbol{k}}, \mathbb{1}\{g_1(\tilde{\boldsymbol{x}}_{\boldsymbol{k}}) \leq 0\})\}_{k=1}^{N}$ and $\{(\tilde{\boldsymbol{x}}_{\boldsymbol{k}}, \mathbb{1}\{g_2(\tilde{\boldsymbol{x}}_{\boldsymbol{k}}) \leq 0\})\}_{k=1}^{N}$ that will be used to train ML models for approximating the nonlinear constraints $g_1(\boldsymbol{x}) \leq 0$ and $g_2(\boldsymbol{x}) \leq 0$.

The sampling is performed in the following steps: (i) boundary sampling, (ii) latin hypercube sampling, (iii) kNN quasi-newton sampling and (iv) OCT-based adaptive sampling. The first 3 sampling steps are the same as the ones used in OCTHaGOn and are described in detail in Sect. 2. Then, sampling step (iv), which is part of our enhancements, is used to adaptively obtain more refined samples in areas where the nonlinear constraints are not approximated well by the ML learners (e.g. near the feasibility boundaries of the constraint). This step is described in detail in Sect. 3.2.

After applying those sampling steps, we can see the samples obtained for the constraints $g_1(\boldsymbol{x}) \leq 0$ and $g_2(\boldsymbol{x}) \leq 0$ in the Figs. 2a and 2b respectively. By examining the samples and the orange feasibility regions of the constraints, we can see that the adaptive sampling procedures we used has helped generate samples near the boundaries of the constraint feasibility regions, which is essential for good ML approximations.

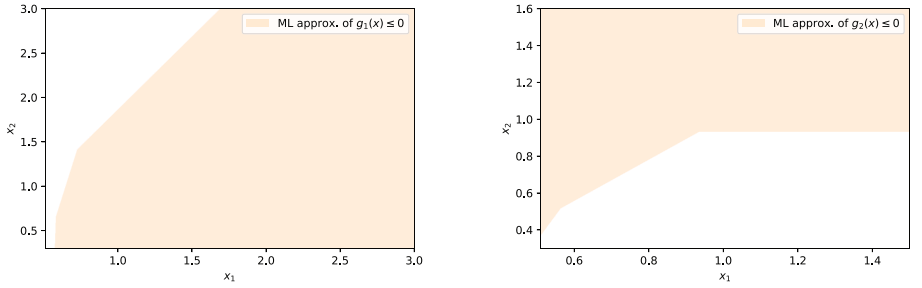
4.3 Model training

Given the samples we obtained from the previous step, we train a number of learners in order to approximate the nonlinear constraints $g_1(\boldsymbol{x}) \leq 0$ and $g_2(\boldsymbol{x}) \leq 0$. The learners are trained on the datasets $D_1 = \{(\tilde{\boldsymbol{x}}_{\boldsymbol{k}}, \mathbb{1}\{g_1(\tilde{\boldsymbol{x}}_{\boldsymbol{k}}) \leq 0\})\}_{k=1}^{N}$ and $D_2 = \{(\tilde{\boldsymbol{x}}_{\boldsymbol{k}}, \mathbb{1}\{g_2(\tilde{\boldsymbol{x}}_{\boldsymbol{k}}) \leq 0\})\}_{k=1}^{N}$ that contain the samples for the constraints $g_1(\boldsymbol{x}) \leq 0$ and $g_2(\boldsymbol{x}) \leq 0$ respectively. For each one of those datasets, we separated the samples into a training and a validation set. We then used



(a) Samples of $g_1(\boldsymbol{x}) \leq 0$ (b) Samples of $g_2(\boldsymbol{x}) \leq 0$

Fig. 2 Feasible region and samples of the nonlinear constraints



(a) MLP approximation of $g_1(\boldsymbol{x}) \leq 0$ (b) OCT approximation of $g_2(\boldsymbol{x}) \leq 0$

Fig. 3 ML approximation of the nonlinear constraints

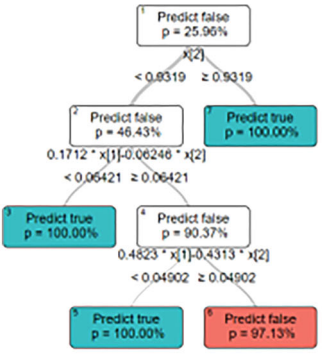
the training sets to train a multi-layer perceptron (MLP), a support vector machine (SVM), a gradient boosted machine (GBM) and Hyperplane-based decision tree (OCT). Then, we measured the accuracy of those models in the respective validation sets and for each of the 2 constraints, we picked the learner that demonstrated the highest accuracy.

In the case of the constraint $g_1(\boldsymbol{x}) \leq 0$, the best learner was the MLP with an accuracy of 0.97, whereas in the case of the constraint $g_2(\boldsymbol{x}) \leq 0$, the best learner was the OCT with an accuracy of 0.99. The ML approximation for the constraints involving g_1 and g_2 are shown in Figs. 3a and 3b respectively.

4.4 MIO representation

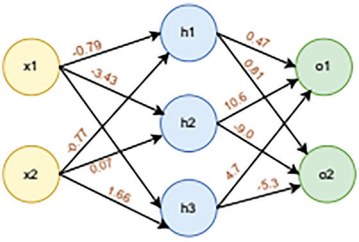
In this step, we represent the trained learners using an MIO formulation as described in Sect. 3.1. In particular, the OCT approximator of the constraint $g_2(\boldsymbol{x}) \leq 0$ can be shown in Fig. 4 in the form of a hyperplane-based decision tree. The tree consists of 3 non-terminal and 4 terminal (leaf) nodes. The leaf nodes that correspond to feasible regions are the ones shown in blue.

In order to represent this decision tree using an MIO formulation, we introduce 4 binary auxiliary variables (z_1, z_2, z_3, z_4), 1 for each leaf. Each auxiliary variable becomes active if and only if $\boldsymbol{x}$ lies in the corresponding leaf. Then, we use a big-M formulation to encode the output of the Decision Tree, as shown next to Fig. 4. Note that in the resulting formulation, y_{OCT} is the output of the decision tree and ϵ is a small positive constant used to model



$$
\begin{aligned}
y_{OCT} &= z_1 + z_2 + z_3, \\
1 &= z_1 + z_2 + z_3 + z_4, \\
x_2 &\geq 0.9319 - M(1 - z_1), \\
x_2 &\leq 0.9319 + M(1 - z_2) - \epsilon, \\
0.1712x_1 - 0.06246x_2 &\leq 0.06421 + M(1 - z_2) - \epsilon, \\
x_2 &\leq 0.9319 + M(1 - z_3) - \epsilon, \\
0.1712x_1 - 0.06246x_2 &\geq 0.06421 - M(1 - z_3), \\
0.4823x_1 - 0.4313x_2 &\leq 0.04902 + M(1 - z_3) - \epsilon, \\
x_2 &\leq 0.9319 + M(1 - z_4) - \epsilon, \\
0.1712x_1 - 0.06246x_2 &\geq 0.06421 - M(1 - z_4), \\
0.4823x_1 - 0.4313x_2 &\geq 0.04902 + M(1 - z_4),
\end{aligned}
$$

Fig. 4 OCT approximator of $\mathbf{g_2}(\boldsymbol{x}) \leq \mathbf{0}$



$$
\begin{aligned}
y_{MLP} &= \boldsymbol{a_3}^T \cdot [-1,\ 1], \\
\boldsymbol{a_3} &\geq \mathbf{0}, \\
\boldsymbol{a_3} &\geq \boldsymbol{W_2 a_2} + \boldsymbol{b_1}, \\
\boldsymbol{a_3} &\leq M\mathbf{v_2}, \\
\boldsymbol{a_3} &\leq \boldsymbol{W_2 a_2} + \boldsymbol{b_2} - M(1 - \boldsymbol{v_2}), \\
\boldsymbol{a_2} &= \boldsymbol{W_1 x} + \boldsymbol{b_1}, \\
\boldsymbol{a_2} &\in \mathbb{R}^3,\ \boldsymbol{a_2} \in \mathbb{R}^3,\ \boldsymbol{v_2} \in \{0,1\}^3.
\end{aligned} \tag{14}
$$

Fig. 5 MLP approximator of $\mathbf{g_1}(\boldsymbol{x}) \leq \mathbf{0}$

strict inequalities. Then, given the MIO representation of the decision tree of Fig. 4, we can approximate the constraint $g_2(\boldsymbol{x}) \leq 0$ with the constraint $y_{OCT} \geq 0.5$.

Next, the trained MLP that approximates the constraint $g_1(\boldsymbol{x}) \leq 0$ is shown in Fig. 5. This MLP consists of 2 input nodes, 3 hidden nodes with ReLU activations and 2 output nodes. It was trained using Negative Log Likelihood loss in a binary classification task, so that it predicts that the constraint is feasible whenever $o_2 \geq o_1$ (i.e. o_1 and o_2 are the output nodes shown in Fig. 5). Note here that although we have a binary classification task, we have used 2 output nodes instead of 1, where an output pair $(o_1, o_2) = (1, 0)$ corresponds to an infeasible input $\boldsymbol{x}$, while an output pair $(o_1, o_2) = (0, 1)$ corresponds to a feasible input $\boldsymbol{x}$.

In order to model the MLP using MIO, we first use a vector of continuous variables $\boldsymbol{a}_i$ to represent the input of the i-th layer of the MLP (i.e. the input of the first hidden layer is denoted as $\boldsymbol{a}_2$, the input of the next layer as $\boldsymbol{a}_3$ etc). Then, for the hidden layer, we additionally use a vector of binary variables $\boldsymbol{v_2}$ to model the ReLU activation. The resulting big-M formulation is described in Eq. (14), where $\boldsymbol{W_1}$, $\boldsymbol{W_2}$ are the weight matrices and $\boldsymbol{b_1}$, $\boldsymbol{b_2}$ are the bias matrices of the trained MLP. Using this notation, the constraint $g_1(\boldsymbol{x}) \leq 0$ can be approximated with the constraint $y_{MLP} \geq 0$, where y_{MLP} is given in Eq. (14).

Then, we combine the MIO formulations of the OCT and the MLP into a unified MIO formulation. In this formulation, we also include the linear constraints of the original problem, and this way we end up with an MIO approximation of the problem. In Fig. 6 we can see the resulting MIO approximation of the feasible space against the actual feasible space of the original problem. We can see that the non-convexity of the feasible region is captured by the MIO formulation, and thus by optimizing over this MIO approximation, we are (approximately) solving the original problem.

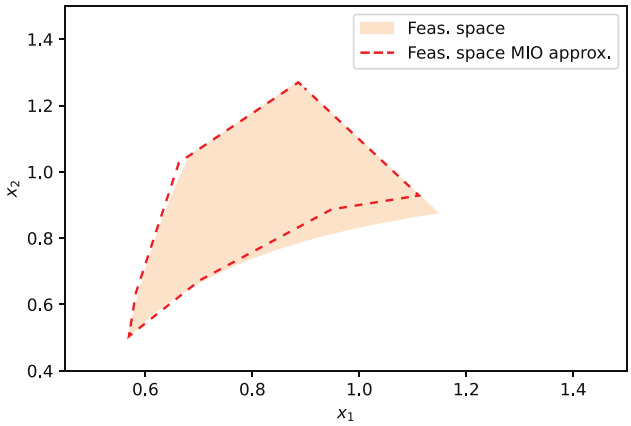


Fig. 6 MIO approximation of feasible space

4.5 Robustness

Constructing a data-driven MIO aproximation of the original problem is not always accurate, in part due to the uncertainty introduced by the sampling process. This uncertainty be partially accounted for by introducing a level of robustness into the MIO formulation. For example, following the steps we described in Sect. 3.4, we can rewrite the MIO representation of the OCT-H of Fig. 4 in the following robust way:

$$
\begin{aligned}
y_{OCT} &= z_1 + z_2 + z_3, \\
1 &= z_1 + z_2 + z_3 + z_4, \\
x_2 - \rho||x_2||_q &\geq 0.9319 - M(1 - z_1), \\
x_2 + \rho||x_2||_q &\leq 0.9319 + M(1 - z_2) - \epsilon, \\
0.1712x_1 - 0.06246x_2 + \rho||0.1712x_1 - 0.06246x_2||_q &\leq 0.06421 + M(1 - z_2) - \epsilon, \\
x_2 + \rho||x_2||_q &\leq 0.9319 + M(1 - z_3) - \epsilon, \\
0.1712x_1 - 0.06246x_2 - \rho||0.1712x_1 - 0.06246x_2||_q &\geq 0.06421 - M(1 - z_3), \\
0.4823x_1 - 0.4313x_2 + \rho||0.4823x_1 - 0.4313x_2||_q &\leq 0.04902 + M(1 - z_3) - \epsilon, \\
x_2 + \rho||x_2||_q &\leq 0.9319 + M(1 - z_4) - \epsilon, \\
0.1712x_1 - 0.06246x_2 - \rho||0.1712x_1 - 0.06246x_2||_q &\geq 0.06421 - M(1 - z_4), \\
0.4823x_1 - 0.4313x_2 - \rho||0.4823x_1 - 0.4313x_2||_q &\geq 0.04902 + M(1 - z_4).
\end{aligned}
\tag{15}
$$

Next, depending on the value of q (which is determined by the type of uncertainty we are protecting against), we reformulate the norm operators in a tractable way. For instance, if we want to protect against L_1 or L_∞ uncertainty sets, then q takes the value ∞ and 1 respectively, and the robust MIO can be easily reformulated using linear constraints. On the other hand, if we want to protect against L_2 uncertainty, then the resulting MIO is conic quadratic.

Note that the extend to which we include robustness into our MIO approximation is determined by the hyperparameter ρ: a value of $\rho = 0$ indicates that we don't want to consider robustness, while higher values of ρ correspond to bigger uncertainty sets and more conservative robust counterparts. For the purpose of this example, we use a value of $\rho = 0.1$.

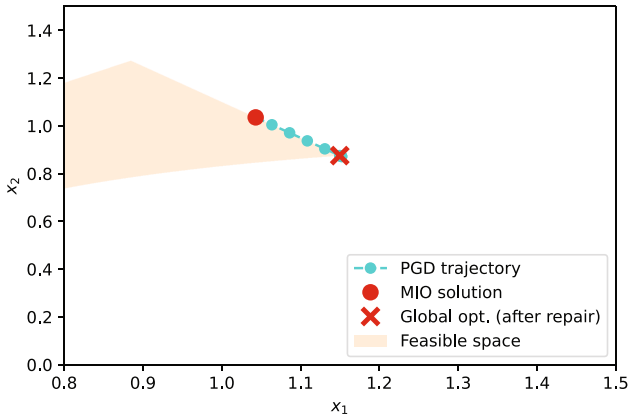


Fig. 7 PGD trajectory

4.6 Solve MIO and improve

After having constructed an MIO approximation of Problem 13, we optimize over this approximation using Gurobi. The initial solution we get is $[x_1, x_2] = [1.108, 0.937]$ with an objective value of -1.108. This initial solution is not optimal, due to the MIO approximation error shown in Fig. 6.

We then proceed to improve the solution using the PGD local-search procedure described in Sect. 2. Note that in this procedure, we also introduce momentum and we conditionally utilize second order information for faster and more accurate convergence. The resulting solution after 10 iterations of the repair procedure is $[x_1, x_2] = [1.1497, 0.875]$ with an objective value of -1.1497, which is the global optimum. The trajectory followed by the PGD iterations can be seen visually in Fig. 7.

5 Computational experiments on benchmarks

In order to test the new framework, which we will refer to as GoML (global optimization using machine learning), we use a number of global optimization problems from the literature. We focus on continuous non-convex optimization problems where the optimization variables are all bounded. The reason we concentrate on problems with bounded variables is that GoML and OCTHaGOn both require variable bounds in order to perform the initial sampling steps. In case variable bounds are not provided, the frameworks try to infer those bounds, but we will only focus on already bounded problems in order to have a more consistent ground of comparison.

5.1 GoML implementation

The GoML framework is implemented in Julia and has been tested on Julia 1.6.2 with an Interpretable AI version of 2.2 and Gurobi 8 as the underlying MIO solver. The implementation of GoML can be found on https://github.com/margaeor/GoML.jl. The code has been built on top of the publicly available OCTHaGOn repository.

5.2 Experimental procedure

In order to perform the experiments, we use 2 different versions of our GoML framework. In the first version (GoML Base), we use the standard convex version of Gurobi to solve the MIO approximation of the original problem. This means that we pass directly to Gurobi the linear constraints (and objectives) of the original problem, and we approximate all the non-linear ones using ML models. This approach is analogous to the one used by OCTHaGOn. In the second version (GoML NonCVX), we also pass the convex constraints and the non-convex quadratic constraints directly to Gurobi, and we use ML to approximate only the non-convex constraints that are not quadratic. The second approach leverages the ability of Gurobi 8.0 to solve non-convex MINLP problems, as long as the non-convex constraints are quadratic. The caveat of this approach is that some of the benchmarks contain only non-convex quadratic constraints and objectives, and thus in those instances, GoML NonCVX offloads all the constraints (and objective) to Gurobi. However, in many of those instances, GoML Base is also able to solve them equally well, as seen in Table 3.

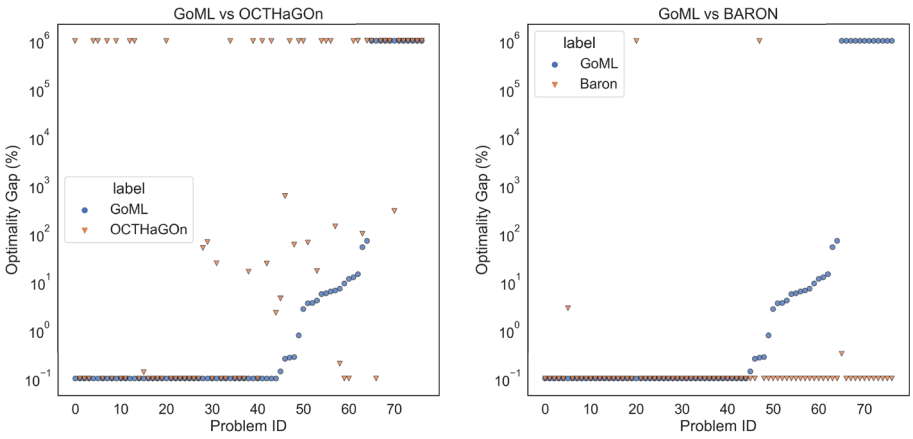
Both versions of GoML have a number of hyperparameters, the most important of which is the robustness radius ρ and the relaxation coefficient λ. In order to determine the best value of those hyperparameters, we perform a grid search. In particular, we run GoML for different values of the hyperparameters and we keep the solution with the best objective value. For the robustness parameter ρ, we use values $\rho = 0$ (i.e. no robustness) and $\rho = 0.01, 0.1, 1$ which represent uncertainty sets of different sizes. For the relaxation coefficient λ, we first experiment with no relaxations (corresponding to $\lambda = \infty$) and then, with varying levels of relaxation penalties ($\lambda = 10^2$ and $\lambda = 10^4$). Finally, for each instance, we also vary the type of sampling we do (i.e. we solve the instance both with and without OCT Sampling).

If we use a naive grid search approach and we solve the problem from scratch for each combination of hyperparameters, then our approach will perform very poorly time-wise. However, we notice that when we change the robustness and relaxation parameters, we don't need to retrain the learners, since both robustness and relaxations are applied to the MIO approximations after the learners are trained. Hence, in our grid search, we first sample and train the ML models, and we then re-embed the models and resolve the MIO for each value of ρ and λ, without having to retrain the models. This observation is really important, since the sampling and training steps take the vast majority of the solution time (i.e. in many cases, more than 95%).

5.3 Results

Using grid-search as mentioned above, we test GoML Base and GoML NonCVX against continuous global optimization problems from MINLPLib [32]. For our benchmarks, we use problems that contain between 1 and 110 variables and between 1 and 88 constraints. We first benchmark our framework (GoML base) against OCTHaGOn, in order to see the effect of our enhancements. Then, we compare against BARON [33], a well-established global optimizer for mixed-integer nonlinear programs. For all methods, we measure the percent optimality gap and the solution time in seconds. We set a pre-specified time limit of 1500s for all solvers, and we run our experiments in a Dell Inspiron Laptop with an 8-core Ryzen 5700U processor clocked at 1.8GHz and 16GB of RAM.

In Figs. 8a and 8b we compare the optimality gaps of GoML Base against OCTHaGOn and BARON respectively among 77 MINLPLib instances. By examining Fig. 8a, we can see that GoML Base has better optimality gaps than OCTHaGOn in 36 out of the 77 instances,



(a) Gap (%) of GoML Base vs OCTHaGOn (b) Gap (%) of GoML Base vs BARON

Fig. 8 Comparison of GoML Base, BARON and OCTHaGOn optimality gaps for each instance. The x-axis represents the instance ID (from 0 to 76) and the y-axis represents the percentage optimality gap. Solutions that are infeasible are assigned a gap of 10^6 in the plot, while solutions that have a gap within 0.1% of the optimal solution are considered optimal. For easier comparison, the instances are sorted from left to right by increasing gap of the GoML framework

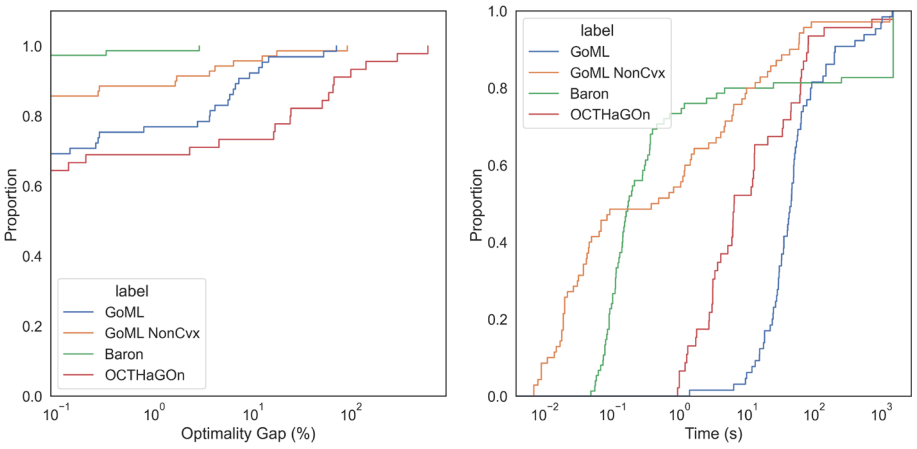
and worse optimality gaps in only 5 of the 77 instances. This means that GoML Base offers solutions that are at least as good as OCTHaGOn in 72 out of the 77 instances, validating that our enhancements are indeed effective.

On the other hand, based on Fig. 8b, BARON is able to solve 73 out of the 77 instances to optimality, which is expected since those problems have long been used as benchmarks in the global optimization literature. However, in 3 of the 77 instances, GoML Base provides solutions with better optimality gaps than BARON. Additionally, in another 4 of the 77 instances, both GoML Base and BARON solve the problem to optimality, but GoML base has better solution times (timing results available in "Appendix B").

Finally, in Figs. 9a and 9b we also show the cumulative performance plots of all the methods in terms of both optimality gap and execution time. By examining the figures, it is evident that GoML Base has generally much better gaps than OCTHaGOn, but worse execution times. This is to be expected, since OTHGaGOn only uses a subset of models and does not perform grid-search. On the other hand, with the exception of BARON, the method that strikes the best balance between optimality gap and time is GoML NonCVX. In particular, in the majority of instances, GoML NoNCVX has better optimality gaps and execution time than both GoML Base and OCTHaGOn. Also, in around 50% of the instances, GoML NonCVX has better execution time than BARON.

5.4 Performance attributions

To better understand the effect of the various enhancements on the overall performance of the algorithm, we ran GoML Base on the 77 MINLPLib instances both with and without the various enhancements (i.e. robustness, OCT sampling and relaxations). Then, we measured



(a) Optimality gap performance plot (b) Time performance plot

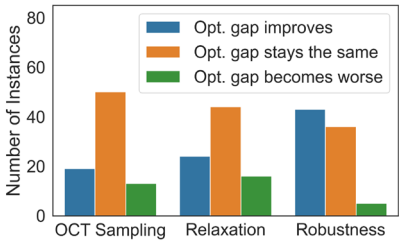
Fig. 9 Cumulative performance plots of the different methods with respect to optimality gap and time. The y-axis represents the proportion of instances where an algorithm has a gap (or execution time) less than the threshold defined by the x-axis

the number of instances in which a particular enhancement improved, kept the same or deteriorated the optimality gap. The result is shown on Fig. 10a. We also tested the performance of the framework by removing one ML model at a time (i.e. MLP, SVM, OCT, GBM). Then, we measured the number of instances in which the inclusion of each specific model improves the optimality and feasibility gap. The result is shown on Fig. 10b. Finally, to better quantify the effect of OCT Sampling, we measured the average percentage constraint violation of the final solution both with and without OCT Sampling. The comparison is shown on Fig. 10c.

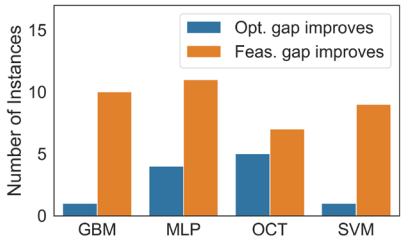
By examining Fig. 10a, we notice that each enhancement improves the optimality gap in at least 20 of the 77 instances. Out of the various enhancements, robustness seems to have the biggest effect on optimality gap, improving the gap in around 40 instances. On the other hand, OCT sampling seems to have the smallest effect on optimality gap. However, as per Fig. 10c, OCT Sampling offers a very substantial improvement in feasibility gap, which is directly related to the fact that OCT sampling helps build accurate constraint approximators through high-quality sampling.

In any case, we should note that as seen on Fig. 10a, there are also a handful of instances where the gap becomes worse as a result of the enhancements. However, this is not problematic, and is precisely the reason why the algorithm involves a grid-search procedure as discussed in Sect. 5.2. Effectively, this procedure applies the framework on each instance with and without the various enhancements, and then keeps the best solution, hence negating the potential adverse effect of specific enhancements in specific instances.

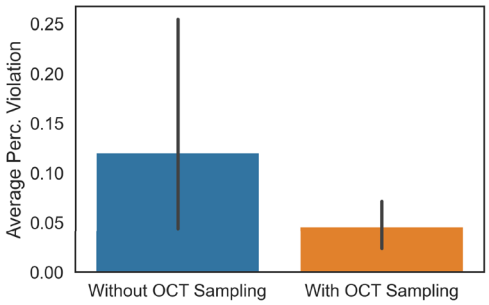
Apart from the enhancements of Fig. 10a, the choice of ML models also has an effect on optimality and feasibility gaps. According to Fig. 10b, MLP and GBM are the most helpful in reducing feasibility gaps, while OCTs and MLPs help the most in reducing optimality gaps. On the other hand, SVM seems to offer the least improvement overall, which is expected since it is the simplest model.



(a) Number of instances in which a particular enhancement improves, keeps the same or deteriorates the optimality gap.



(b) Number of instances in which each ML model improves the optimality and feasibility gaps.



(c) Effect of OCT sampling on average percentage constraint violation of the final solution.

Fig. 10 Effects of the various enhancements and ML models on optimality and feasibility gaps

6 Experiments on real-world and synthetic examples

In this section, we test our approach in some synthetic and real world problems. The purpose of this section is to explore problems that aren't part of the popular MINLP benchmarks of Sect. 5.

6.1 Quadratic and sigmoid

First, we consider the following class of synthetic problems:

$$\begin{aligned} \min_{\boldsymbol{x}} \quad & \boldsymbol{c}^T \boldsymbol{x} \\ \text{s.t.} \quad & \frac{1}{1+\exp\{-Q_i(\boldsymbol{x})\}} \leq 0.5, \quad i = 1, \ldots, \lfloor m/2 \rfloor, \\ & \frac{Q_i(\boldsymbol{x})}{1+\exp\{-Q_i(\boldsymbol{x})\}} \geq -0.5, \quad i = \lfloor m/2 \rfloor + 1, \ldots, m, \\ & x_k \in [\underline{x}_k, \overline{x}_k], \ k \in [n], \\ & \boldsymbol{x} \in \mathbb{R}^n, \end{aligned} \tag{16}$$

Table 1 GoML vs BARON in Quadratic–Sigmoid problems

n	m	GoML Obj	BARON Obj	BARON LB	GoML Opt. Gap	BARON Opt. Gap
10	2	−22.83	**−23.04**	−23.04	0.9%	0
50	4	**−177.27**	−0.225	−189.728	$\leq$ 6.56%	$\geq$ 99.9%
70	2	**−255.85**	−0.5711	−257.285	$\leq$ 0.8%	$\geq$ 99.8%

Bold indicates the solution with the best objective value

where $Q_i(\boldsymbol{x})$ is a quadratic:

$$Q_i(\boldsymbol{x}) = \boldsymbol{x}^T \boldsymbol{A}_i \boldsymbol{x} + \boldsymbol{d}_i^T \boldsymbol{x} + f_i. \tag{17}$$

In this problem, the objective is linear and the constraints involve quadratic and sigmoid functions. Finally, we also impose bounds on the decision variables, forcing them to belong to a particular hyper-rectangle. In order to benchmark our framework in instances of this problem, we first choose a dimension n and a number of nonlinear constraints m. Then, we randomly generate the cost vector $\boldsymbol{c}$ and the quadratic parameters $\boldsymbol{A}_i$, $\boldsymbol{d}_i$ and f_i for each $i = 1, \ldots, m$. We repeat the same process for different values of n and m. Finally, we run the GoML Base framework in the generated instances and we compare against BARON.

In Table 1 we can see the resulting objective values returned by the 2 methods for problem instances of different sizes. We also record the lower bound calculated by BARON during the solution process. Finally, since the optimal solution of the instances is not known a-priori, we use the lower bound returned by BARON and the best feasible solution across the two methods to calculate bounds on the optimality gap of GoML and BARON. Those bounds are recorded in Table 1.

Our experiments show that BARON solves the problem to optimality for small values of n, but performs very poorly for bigger n. For instance, for $n = 70$ and $m = 2$, BARON terminates early with a solution that has an optimality gap of at least 99.8%, while the optimality gap of GoML is at most 0.8%. A similar pattern is seen for $n = 50$ and $m = 4$, where BARON terminates early with an optimality gap of at least 99.9%, while GoML has an optimality gap of at most 6.56%.

6.2 Speed reducer problem

Following [7], we also test the method in the Speed Reducer problem proposed by [34]. This is a real world problem with the goal of designing a gearbox for an aircraft engine under geometrical, structural and manufacturing constraints. The problem is described below:

$$
\begin{aligned}
\min_{\boldsymbol{x}} \quad & 0.7854x_1x_2^2\left(3.3333x_3^2+14.9334x_3-43.0934\right)-1.5079x_1\left(x_6^2+x_7^2\right)+\\
& +7.477\left(x_6^3+x_7^3\right)+0.7854(x_4x_6^2+x_5x_7^2)\\
\text{s.t.} \quad & -27+x_1x_2^2x_3\geq 0,\ -397.5+x_1x_2^2x_3^2\geq 0,\\
& -1.93+\frac{x_2x_6^4x_3}{x_4^3}\geq 0,\ -1.93+\frac{x_2x_7^4x_3}{x_5^3}\geq 0,\\
& 110.0x_6^3-\left(\left(\frac{745x_4}{x_2x_3}\right)^2+16.9\times 10^6\right)^{0.5}\geq 0, \qquad (18)\\
& 85.0x_7^3-\left(\left(\frac{745x_5}{x_2x_3}\right)^2+157.5\times 10^6\right)^{0.5}\geq 0,\\
& 40-x_2x_3\geq 0,\ x_1-5x_2\geq 0,\ 12x_2-x_1\geq 0,\\
& x_4-1.5x_6-1.9\geq 0,\ x_5-1.1x_7-1.9\geq 0,\\
& \boldsymbol{x}\geq[2.6,0.7,17,7.3,7.3,2.9,5],\\
& \boldsymbol{x}\leq[3.6,0.8,28,8.3,8.3,3.9,5.5],\\
& x_3\in\mathbb{Z}. \qquad (19)
\end{aligned}
$$

In Table 2 we compare the solution given by the enhanced GoML framework with the solutions given by OCTHaGOn and IPOPT. The optimal solutions, objectives and timings for OCTHaGOn and IPOPT are all retrieved from [7]. We also compare against the optimum by [35], which according to [7], is the best known optimum in the literature. We observe that the ML-based global optimization methods (OCTHaGOn, GoML) perform very well in this real world problem, being able to find the optimal solution in a reasonable time. Also, OCTHaGOn, GoML and IPOPT all provide a better solution than that of [35].

According to the results, IPOPT solves the problem to optimality, beating OCTHaGOn and GoML with respect to solution time. We should, however, note that as described in [7], in order for IPOPT to be applied to that problem, the integrality constraint of x_3 needs to be relaxed, since IPOPT does not handle integer variables. In this particular problem this was not an issue, since the optimal value returned by IPOPT for x_3 was indeed an integer. However, IPOPT cannot be used for general MINLP problems. On the other hand, the Mixed Integer nature of OCTHaGOn and GoML natively allows enforcing integrality constraints.

We should note that the solutions of Table 2 with $x_6 = 3.3502$ may appear infeasible when plugged to the original problem. This happens because despite the precision of 10^{-4} used in Table 2, the high nonlinearities of constraints (18) and (19) lead to infeasibilities unless very high precision is used. In reality, both GoML and OCTHaGOn return a value

Table 2 Comparison of methods in the speed reducer problem

	x_1	x_2	x_3	x_4	x_5	x_6	x_7	Objective	Time (s)
Lin & Tsai [35]	3.5	0.7	17	7.3	7.7153	3.3503	5.2867	2994.472	476
OCT-HaGOn	3.5	0.7	17	7.3	7.7153	3.3502	5.2867	2994.355	32.6
GoML	3.5	0.7	17	7.3	7.7153	3.3502	5.2867	2994.355	37
IPOPT	3.5	0.7	17.0*	7.3	7.7153	3.3502	5.2867	2994.355	4.2

of $x_6 = 3.35021466$ and $x_7 = 5.28665446$, which make the constraints feasible within a tolerance of 10^{-5} and yield an objective value of 2994.35458.

7 Discussion

In this section, we discuss the limitations of the approach and suggest avenues for future work.

7.1 Contributions

In this work, we provided extensions to the global optimization framework OCTHaGOn, in order to improve feasibility and optimality of the generated solutions. We tested the framework on a mix of 81 Global Optimization instances, with 77 of those being part the standard benchmarking library MINLPLib and the rest being part of real-world and synthetic instances. Our results showed that in the majority of instances, our enhancements improve the optimality gaps and solutions times of OCTHaGOn. We also identified a number of instances where the enhanced framework provides better or faster solutions than BARON.

Overall, we showed that despite its approximate nature, the enhanced framework, is a promising method in finding globally optimal solutions in various types of problems. The framework can potentially be applied in very general global optimization problems, including problems with constraints that are convex, non-convex and constraints with very general primitives, with the basic assumption that the user specifies bounds for the decision variables. Hence, due to its generality the method can be used in problems that are incompatible with traditional global optimizers such as BARON and ANTIGONE.

7.2 Limitations

The enhanced framework demonstrates promise in tackling a variety of global optimization problems, but it is still a work in progress and comes with its own sets of limitations.

Since the method relies on ML-based approximations of the original problem, it does not necessarily produce globally optimal solutions. The PGD step at the end of the method generally helps in finding high-quality solutions, but without offering any optimality guarantees such as the ones provided by BARON. This means that although the framework performs well in a wide range of problems, it is actually a heuristic method without any optimality guarantees. Additionally, the PGD repair step assumes that the nonlinear constraints support automatic differentiation (AD). Although this is a soft assumption to make, it may not be true for certain types of constraints. In some cases, the lack of AD can be addressed by approximately evaluating the gradients (i.e. through finite differencing). However, this approach also has its limitations, since it may produce very noisy estimates of the gradient.

Another limitation of the method has to do with its stated assumptions. The framework assumes that the decision variables involved in nonlinear constraints have prespecified bounds, where in practice bounds may not be available. In case where bounds are not specified, the method attempts to compute them through an optimization process, although this is not always effective, since user-specified bounds usually lead to much more precise solutions. The method also assumes that the nonlinear constraints, are fast to evaluate. If this is not the case, such as in implicit or simulation-based constraints, then it may be difficult

for the framework to produce high-quality and efficient solutions, since our solution process inherently obtains many samples from each nonlinear constraint.

Another consideration about the method is computational speed. Although the framework is relatively fast for small and medium-sized problems, the computational time can rise significantly with the number of variables and nonlinear constraints. The main bottleneck of the method is the training of the ML models, and particularly of the hyperplane-based decision trees (OCT-H), which are used in the sampling and approximation phases of the framework. To partially account for this problem, we restrict the use of OCT-Hs to smaller problems, although the ML training time still rises linearly with the number of nonlinear constraints. Another speed-related consideration has to do with the complexity of the MIO approximations. For small and medium-sized problems, the complexity of the MIO approximation is usually very small compared to the capabilities of commercial solvers like Gurobi and CPLEX. However, an increase in the number of variables accompanied with an increase in model complexity (e.g. deeper trees and more layers for MLPs) can significantly affect MIO solution times.

8 Conclusion

In this work, we implemented a range of enhancements to improve the OCTHaGOn [7] global optimization framework. We used ML-based sampling, Robust Optimization, and other techniques to improve the optimality and feasibility gaps of the solutions generated by OCTHaGOn. We then demonstrated the effect of our enhancements through a range of global optimization benchmarks. We compared the framework against the commercial optimizer BARON, and we showed improved solution times and optimality gaps in a subset of problems. More concretely, we showed that in the majority of test instances, the enhancements improve the optimality gaps and solution times of OCTHaGOn. We also showed that in 9 instances (i.e., 7 MINLP and 2 synthetic instances), the enhanced framework yields better or faster solutions than BARON.

The overall method is a general way of addressing global optimization problems, that is generally new in the global optimization literature. It can handle constraints that are convex, non-convex or consist of very general mathematical primitives. The method only requires bounds of the decision variables involved in the nonlinear constraints. Although the method is still new, it can potentially be used in a number of different applications, especially in areas where typical global optimizers cannot be used.

Appendix A Enhancement details

In this appendix we provide supplementary information about the various enhancements. We discuss about the training and MIO-representation of the different ML models. We also provide a detailed description of OCT Sampling.

A.1 Support vector machines

Support vector machines are ML models that use a suitable hyperplane to make predictions, either for classification [23] or regression [24]. In this work, we use linear SVRs and SVCs to approximate nonlinear objectives and constraints respectively.

In order to approximate a nonlinear objective with SVMs, we train a support vector regressor (SVR) on regression samples of the objective. Training an SVR is done in a very similar way as linear regression, but in a way that we penalize residuals greater than an ϵ threshold [19, 24]. Then, we can embed the SVR predictions using the following linear constraint:

$$y_{SVR} = \beta_0 + \boldsymbol{x}^T \boldsymbol{\beta}. \tag{A1}$$

On the other hand, in order to approximate a nonlinear constraint, we train a support vector classifier (SVC) on feasibility samples of the constraint. The training process is very similar to the SVR, but in this case y_i are labels. Then, we can approximate the nonlinear constraint with the following constraint:

$$\beta_0 + \boldsymbol{x}^T \boldsymbol{\beta} \geq 0, \tag{A2}$$

where $\boldsymbol{\beta}$, β_0 are the trained model parameters of the SVM.

A.2 Decision trees

For the purpose of this work, we will model the generalized version of decision trees which includes hyperplane splits [8], since this version can also capture the standard CART trees with parallel splits. In particular, a decision tree can be modeled as a binary tree with non-terminal (intermediate) and terminal (leaf) nodes. Each non-terminal node N_i represents a split of the form $\boldsymbol{a}_i^T \boldsymbol{x} \leq b_i$, and each leaf node L_i is associated with a prediction p_i, where $\boldsymbol{a}_i, b_i, p_i$ are model parameters learned through training. Then, given an input vector $\boldsymbol{x}$, we can make predictions as follows: starting from the root N_1, we check whether $\boldsymbol{a}_1^T \boldsymbol{x} \leq b_1$. If the condition is satisfied, then we proceed to the left child of the root. Otherwise, we proceed to the right child. We recursively repeat this process by checking the split condition of the nodes and continuing traversing the tree until we reach a leaf node. When a leaf node L_i is reached, we output the prediction p_i. If we now use $L(L_i)$ and $R(L_i)$ to denote the set of non-terminal nodes for which leaf L_i is contained in their left and right subtree respectively, then each leaf L_i is described by the following polyhedron:

$$\mathcal{P}_i = \Big(\bigcap_{j \in L(L_i)} \{\boldsymbol{x} \in \mathbb{R}^n : \boldsymbol{a}_j^T \boldsymbol{x} \leq b_j\} \Big) \bigcap \Big(\bigcap_{j \in R(L_i)} \{\boldsymbol{x} \in \mathbb{R}^n : \boldsymbol{a}_j^T \boldsymbol{x} > b_j\} \Big). \tag{A3}$$

Then, the decision tree predicts p_i if and only if $\boldsymbol{x} \in \mathcal{P}_i$. Note here that the polyhedra $\mathcal{P}_1, \mathcal{P}_2, \ldots$ are disjoint and their union MIO representable (i.e. can be represented by a set of linear MI constraints). Additionally, we should mention that this representation can also be used to describe decision trees with parallel splits (i.e., CART). In particular, if all the vectors $\boldsymbol{a}_i$ are binary with $\sum_j a_i^{(j)} = 1$, then all the splits in the tree are done in a single feature at a time, and the polyhedrons $\mathcal{P}_i$ are hyper-rectangles.

Let's now use $\mathcal{L}$ to denote the set of leaves. If we want to embed the tree predictions into our MIO under a regression setting, then we can use the following set of constraints to do so:

$$
\begin{aligned}
&\sum_{i=1}^{|\mathcal{L}|} z_i p_i = y_{DT}, \\
&\sum_{i=1}^{|\mathcal{L}|} z_i = 1, \\
&\boldsymbol{a}_j^T \boldsymbol{x} \le b_j + M(1 - z_i), \quad \forall i \in \mathcal{L}, j \in L(L_i), \\
&\boldsymbol{a}_j^T \boldsymbol{x} \ge b_j - M(1 - z_i) + \epsilon, \quad \forall i \in \mathcal{L}, j \in R(L_i),
\end{aligned} \tag{A4}
$$

where y_{DT} represents the output of the decision tree and $\boldsymbol{z} \in \{0, 1\}^{|\mathcal{L}|}$ is a vector of binary variables.

On the other hand, if we want to approximate a nonlinear constraint, we first train a classification DT on samples of the constraint, and we then impose the following additional constraint:

$$
y_{DT} \ge 0.5, \tag{A5}
$$

which ensures that we will land on a feasible leaf.

A.3 Gradient boosted trees

In order to represent a gradient boosted tree (GBT) using an MIO formulation, we use the formulations from [17] with the coupling constraints from [27]. In particular, let's assume that we have a trained Gradient Boosted Tree (GBT) with T trees. Our goal is to express the output of the GBT as a Mixed-Integer function of the input variable $\boldsymbol{x}$. We will assume that $x_i \in [v_i^L, v_i^U]$. For each variable x_i, we order all the possible breakpoints of the GBT ensemble: $u_i^L = u_{i,0} < u_{i,1} < \cdots < u_{i,m_i} < u_{i,m_i+1} = u_i^U$. We use binary variable $y_{i,j}$ to model whether $x_i < u_{i,j}$ for $i \in [n]$ and $j \in [m_i]$. We also use binary variable $z_{t,l}$ to model whether tree $t \in \{1, \ldots, T\}$ selects leaf $\ell \in \textbf{leaves}(t)$ or not. We also use the following notation:

- $splits(t)$ is the set of split nodes of tree t.
- $left(s)$ is the set of leaf nodes of the left subtree of split node s.
- $right(s)$ is the set of leaf nodes of the right subtree of split node s.
- $i(s) \in \{1, \ldots, n\}$ is the index of the variable that participates in split s.
- $j(s) \in \{1, \ldots, m_{i(s)}\}$ is the index of the breakpoint we split when at node s.
- a_t is the weight of each tree in the ensemble.
- $p_{t,\ell}$ is the prediction at leaf ℓ of tree t.

Then, following [27], we can represent the GBT ensemble with the following formulation:

$$
\begin{aligned}
y_{GBT} = &\sum_{t=1}^{T} \sum_{\ell \in \textbf{leaves}(t)} a_t \cdot p_{t,\ell} \cdot z_{t,\ell} \\
&\sum_{\ell \in \textbf{leaves}(t)} z_{t,\ell} = 1, && \forall t \in \{1, \ldots, T\}, \\
&\sum_{\ell \in \textbf{left}(s)} z_{t,\ell} \le y_{i(s),j(s)}, && \forall t \in \{1, \ldots, T\}, s \in \textbf{splits}(t), \\
&\sum_{\ell \in \textbf{right}(s)} z_{t,\ell} \le 1 - y_{i(s),j(s)}, && \forall t \in \{1, \ldots, T\}, s \in \textbf{splits}(t),
\end{aligned}
$$

$$
\begin{aligned}
& y_{i,j} \leq y_{i,j+1}, && \forall i \in \mathcal{N},\ j \in \{1,\ldots,m_i-1\},\\
& x_i \geq v_{i,0} + \sum_{j=1}^{m_i} (v_{i,j} - v_{i,j-1})(1 - y_{i,j}), && \forall i \in \{1,\ldots,n\},\\
& x_i \leq v_{i,m_i+1} + \sum_{j=1}^{m_i} (v_{i,j} - v_{i,j+1}) y_{i,j}, && \forall i \in \{1,\ldots,n\},\\
& y_{i,j} \in \{0,1\}, && \forall i \in \{1,\ldots,n\},\ j \in \{1,\ldots,m_i\},\\
& z_{t,\ell} \geq 0, && \forall t \in \{1,\ldots,T\},\ \ell \in \textbf{leaves}(t).
\end{aligned}
$$

This formulation avoids big-Ms and ϵ constants which are present in the standard decision tree formulation (Eq. (A4)). It should be noted that for approximating a nonlinear objective, the formulation can be used as-is, while if we want to approximate a nonlinear constraint, we also need to add the feasibility constraint:

$$y_{GBT} \geq 0.5. \tag{A6}$$

A.4 Neural networks

In this work, we use multi-layer perceptrons (MLPs) with ReLU activations. Such networks consist of an input layer, $L-2$ hidden layers with ReLU activations and an output layer. Following [19], if we define N^l as the set of nodes of l-th hidden layer of the network, then the value v_i^l of each node $i \in N^l$ is calculated as a weighted average of the node values of the previous layer. The result is passed through a ReLU activation yielding the following formula:

$$v_i^l = \max\left\{0,\quad \beta_{i0}^l + \sum_{j \in N^{l-1}} \beta_{ij}^l v_j^{l-1}\right\}, \tag{A7}$$

where β_i^l is the vector of coefficients for node i in layer l retrieved after training. The benefit of such models is that due to their nonlinearities, they can be used to model very complex nonlinear constraints and objectives compared to linear models.

For our use-case, we will only consider MLPs with 1 output neuron. In particular, for the regression task, we train an MLP regressor on the dataset $D_R = \{(\tilde{\boldsymbol{x}}_{\boldsymbol{k}}, f(\tilde{\boldsymbol{x}}_{\boldsymbol{k}}))\}_{k=1}^n$ using Mean Squared Error (MSE) loss, and we then use the following constraints to represent the output of the MLP:

$$
\begin{aligned}
y_{MLP} =& \beta_{00}^L + \sum_{j \in N^{L-1}} \beta_{0j}^L v_j^{L-1},\\
& u_i^l \geq \beta_{i0}^l + \sum_{j \in N^{l-1}} \beta_{ij}^l v_j^{l-1}, && \forall l = \{2,\ldots,L-1\},\ i \in N^l,\\
& u_i^l \leq \beta_{i0}^l + \sum_{j \in N^{l-1}} \beta_{ij}^l v_j^{l-1} - M(1 - z_{il}), && \forall l = \{2,\ldots,L-1\},\ i \in N^l,\\
& u_i^l \leq M z_{il}, && \forall l = \{2,\ldots,L-1\},\ i \in N^l,\\
& u_i^l \geq 0, && \forall l = \{2,\ldots,L-1\},\ i \in N^l,\\
& u_i^1 = x_i, && \forall i \in [n],\\
& z_{il} \in \{0,1\},
\end{aligned}
\tag{A8}
$$

where x_i is the input variable and y_{MLP} is the model's output, which is an approximator of the objective $f(\boldsymbol{x})$. Here, we rely on a big-M formulation to model the ReLU activations, which can be tightened through an appropriate choice of M [19].

For classification, we train an MLP on the dataset $D_C = \{(\widetilde{\boldsymbol{x}}_{\boldsymbol{k}}, \mathbb{1}\{g(\widetilde{\boldsymbol{x}}_{\boldsymbol{k}}) \leq 0\})\}_{k=1}^{N}$ using binary cross-entropy loss and a sigmoid activation on the output node. Then, we approximate the constraint $g(\boldsymbol{x}) \leq 0$ using the formulation (A8), but with the additional constraint $y_{MLP} \geq 0$. This new constraint is used to ensure that the output logit is positive, which corresponds to an output probability greater than 0.5.

A.5 OCT sampling

Below we present the detailed steps that we follow during OCT sampling. Let's assume that we want to approximate the constraint $g(\boldsymbol{x}) \leq 0$ and we already have samples $D = \{(\widetilde{\boldsymbol{x}}_{\boldsymbol{k}}, \mathbb{1}\{g(\widetilde{\boldsymbol{x}}_{\boldsymbol{k}}) \leq 0\})\}_{k=1}^{N}$ for the feasibility of that constraint. The goal of OCT sampling is to resample parts of the constraint which are difficult to approximate. In order to do that, we will use hyperplane-based decision trees (OCT-H, [8, 9]) which are one of the types of learners we use for constraint approximations.

The procedure we follow is the following:

1. **Learner training:** We generate K random subsets $D_1, \ldots, D_K \subset D$ of the dataset D, with a fixed size $|D_i| = C, \ \forall i \in [K]$. We then train one OCT-H learner T_i on each dataset D_i.
2. **Identify ambiguous samples:** In this step, we identify points $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots$ for which there is a high prediction discordance between the learners $T_1, \ldots, T_K$. The goal of identifying such points is that those points are indicators of areas where there is a poor generalization from our learners, and thus areas that are worth resampling. In order to find such points, let's use $T_i(\boldsymbol{x})$ to denote the binary prediction of the i-th learner given an input vector $\boldsymbol{x}$. We then define the following quantities:

$$\begin{aligned} P(\boldsymbol{x}) &= |\{i \in [K] : T_i(\boldsymbol{x}) = 1\}|, \\ N(\boldsymbol{x}) &= |\{i \in [K] : T_i(\boldsymbol{x}) = 0\}|. \end{aligned} \tag{A9}$$

   Then, we find a subset of points of D for which there is a high discordance between the predictions of $T_1, \ldots, T_K$. This set of points S is defined as follows.:

$$S = \{\boldsymbol{x} \in D : |P(\boldsymbol{x}) - N(\boldsymbol{x})| \leq K\tau\}, \tag{A10}$$

   where $\tau \in [0, 1]$ is a threshold that determines the target level of predictor discordance. The higher the value of τ, the less the level of discordance is needed for a point $\boldsymbol{x}$ to be included in S.
3. **Identify ambiguous polyhedra:** The goal of this step is to identify polyhedral regions where the points of S reside. As we discussed in "Appendix A.2", in a hyperplane-based decision tree T_i, every leaf $L_j^{(i)} \in \mathcal{L}_i$ is represented by a polyhedron $\mathcal{P}_j^{(i)}$, while the polyhedra of the different leaves are disjoint. Hence, the decision tree T_i assigns every point $\boldsymbol{x} \in \mathbb{R}^n$ to exactly 1 leaf-polyhedron, which we will denote as $\mathcal{P}^{(i)}(\boldsymbol{x})$. Then, for every point $\boldsymbol{x} \in S$, we take the intersection $\mathcal{P}(\boldsymbol{x})$ of those leaf polyhedras of the different trees $T_1, \ldots, T_K$, where:

$$\mathcal{P}(\boldsymbol{x}) = \bigcap_{i=1}^{K} \mathcal{P}^{(i)}(\boldsymbol{x}). \tag{A11}$$

Note that $\mathcal{P}(\boldsymbol{x})$ is a polyhedron as an intersection of polyhedras. We then create the set of polyhedras $\mathcal{C} = \{\mathcal{P}(\boldsymbol{x}) : \boldsymbol{x} \in S\}$, which has the following property:

$$|P(\boldsymbol{x}) - N(\boldsymbol{x})| \leq K\tau, \quad \forall \boldsymbol{x} \in \mathcal{P},\ \mathcal{P} \in \mathcal{C}. \tag{A12}$$

This means that if we pick any point $\boldsymbol{x}$ from any polyhedron $\mathcal{P} \in \mathcal{C}$, then the predictions of the trees $T_1, \ldots T_K$ for that point are guaranteed to have a particular level of disagreement.

4. **Sample ambiguous polyhedra:** The goal of this step is to generate samples from the interior of the polyhedra $\mathcal{P} \in \mathcal{C}$. As we mentioned before, those polyhedra represent areas where there is a high level of disagreement between the predictors $T_1, \ldots, T_K$. In order to sample those polyhedra, we will use the hit-and-run algorithm [29] which is a Markov Chain Monte Carlo (MCMC) method for generating samples in the interior of a convex body. In particular, given a non-empty polyhedron $\mathcal{P} = \{\boldsymbol{x} \in \mathbb{R}^n : \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}\}$, we can generate samples on its interior using the following procedure:

   (i) Pick a starting point $\boldsymbol{x}_0 \in \mathbb{R}^n$ which lies in the interior of the polyhedron $\mathcal{P}$.
   (ii) Generate a unit random direction $\boldsymbol{u} \in \mathbb{R}^n$
   (iii) Calculate the minimum and maximum values of $\lambda \in \mathbb{R}$ such that $\boldsymbol{x}_0 + \lambda \boldsymbol{u} \in \mathcal{P}$. Those values $\lambda_{\min}$ and $\lambda_{\max}$ can be calculated in close form by requring that $\lambda \boldsymbol{u} \leq \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}$. Note that for general polyhedras, it can happen that $\lambda_{\max} = +\infty$ or $\lambda_{\min} = -\infty$, but the polyhedras we examine are all bounded, so $\lambda_{\max}$ and $\lambda_{\min}$ are finite.
   (iv) Pick a λ_* uniformly at random from the set $[\lambda_{\min}, \lambda_{\max}]$ and generate sample $\boldsymbol{x}_* = \boldsymbol{x}_0 + \lambda_* \boldsymbol{u}$.
   (v) Repeat the procedure from step (ii) using $\boldsymbol{x}_*$ as the new starting point. Terminate whenever we generate the required number of samples.

   We then follow this procedure to generate samples for all polyhedra $\mathcal{P} \in \mathcal{C}$. If the set $\mathcal{P} \in \mathcal{C}$ contains 2 polyhedra with the same representation more than once, then we only sample one of them.

Appendix B Computational results

In this appendix, we are presenting more in-depth computational results on the MINLPLib instances, and we are also testing the method against piece-wise linear constraint approximators introduced into Gurobi 10.0.

B.1 MINLPLib results

In Table 3, we can see the detailed comparison of GoML Base, GoML NonCVX, OCTHaGOn and BARON on the MINLPlib benchmarks. For every example, we report the MINLP instance name, the number of variables, the optimality gap (in percentage) and the time (in seconds).

When the optimality gap is below 0.1%, we consider the solution optimal and we record "GOpt" in the table. In the case of GoML NonCVX, we use an asterisk $*$ to label the instances where the problems are non-convex quadratic and are thus solved in their entirety by Gurobi Non-Convex (i.e. our framework does not approximate any constraint in these instances).

Also, when a method fails to produce a feasible solution to the problem, we record "Inf" in the respective columns.

Table 3 GoML versus BARON versus OCTHaGOn

Name	# vars	GoML base		GoML NonCVX		BARON		OCTHaGOn	
		Gap (%)	Time (s)	Gap (%)	Time (s)	Gap (%)	Time (s)	Gap (%)	Time (s)
ex4_1_2	1	0.14	1.4	**GOpt**	0.8	**GOpt**	**0.4**	4.58	1
ex4_1_3	1	**GOpt**	10.1	**GOpt**	0.7	**GOpt**	**0.1**	2.31	1.3
prob06	1	**GOpt**	28.7	**GOpt***	0.5	**GOpt**	**0.1**	Inf	Inf
ex4_1_6	1	0.26	13.4	6.46	4.7	**GOpt**	**0.2**	607.74	1.3
ex4_1_7	1	**GOpt**	15.7	**GOpt**	4.8	**GOpt**	**0.1**	16.5	1.2
ex4_1_1	1	6.73	63.8	**GOpt**	59.6	**GOpt**	**1.2**	143	11.7
st_e24	2	**GOpt**	50.2	**GOpt***	**0**	**GOpt**	0.1	**GOpt**	3.2
st_e09	2	**GOpt**	35.5	**GOpt***	**0**	**GOpt**	0.1	0.14	3.7
st_ht	2	9.38	55.1	**GOpt***	**0**	**GOpt**	0.1	**GOpt**	3.1
ex4_1_8	2	**GOpt**	17.9	**GOpt**	13.4	**GOpt**	**0.1**	**GOpt**	3.1
st_e19	2	**GOpt**	40.9	**GOpt**	2.8	**GOpt**	**0.2**	**GOpt**	3.4
st_e17	2	**GOpt**	405.8	17.73	1447.1	**GOpt**	**0.1**	24.16	1
st_e23	2	7.31	47.2	**GOpt***	**0**	**GOpt**	0.1	0.2	3.2
st_e01	2	**GOpt**	28.1	**GOpt***	**0**	**GOpt**	**0**	**GOpt**	0.9
ex4_1_9	2	**GOpt**	30.3	**GOpt**	6.5	**GOpt**	**0.1**	**GOpt**	1.8
st_e08	2	**GOpt**	33.1	**GOpt***	**0**	**GOpt**	0.1	**GOpt**	1.8
st_e22	2	**GOpt**	30.7	**GOpt***	**0**	**GOpt**	0.1	**GOpt**	2.8
st_e26	2	**GOpt**	52.1	**GOpt***	**0**	**GOpt**	0.1	**GOpt**	2.8
st_e18	2	**GOpt**	42.6	**GOpt***	**0**	**GOpt**	0.1	Inf	Inf

Table 3 continued

Name	# vars	GoML base		GoML NonCVX		BARON		OCTHaGOn	
		Gap (%)	Time (s)	Gap (%)	Time (s)	Gap (%)	Time (s)	Gap (%)	Time (s)
ex6_2_8	3	53.01	64.6	92.25	9.6	**GOpt**	**2.6**	100.45	6.2
st_e11	3	**GOpt**	29.9	**GOpt**	1.1	**GOpt**	**0.1**	Inf	Inf
st_cqpjk2	3	**GOpt**	69.5	**GOpt***	**0**	**GOpt**	0.1	**GOpt**	6.3
st_e02	3	**GOpt**	58.1	**GOpt***	**0**	**GOpt**	0.1	Inf	Inf
st_bpv1	4	**GOpt**	18.5	**GOpt***	**0**	**GOpt**	0.1	**GOpt**	13.1
sample	4	Inf	Inf	Inf	Inf	**GOpt**	**0.3**	Inf	Inf
ex6_2_9	4	3.65	15.6	3.65	**10.1**	**GOpt**	1501.5	66.21	13.2
st_e12	4	**GOpt**	27.1	**GOpt**	1.1	**GOpt**	**0.1**	**GOpt**	6.1
st_e04	4	2.76	25.2	**GOpt**	20.6	**GOpt**	**0.2**	Inf	Inf
ex6_2_14	4	**GOpt**	12.1	**GOpt**	**1.2**	Inf	Inf	Inf	Inf
ex6_2_12	4	4.16	9.6	4.16	8.5	**GOpt**	**4.7**	17	13.1
st_bpv2	4	**GOpt**	24.7	**GOpt***	**0**	**GOpt**	0.1	**GOpt**	6.5
st_e41	4	**GOpt**	6.5	**GOpt**	4.2	**GOpt**	**0.1**	**GOpt**	12.7
ex3_1_2	5	Inf	Inf	**GOpt***	**0**	**GOpt**	0.1	**GOpt**	12.4
ex2_1_1	5	**GOpt**	47.1	**GOpt***	**0**	**GOpt**	1501.4	67.65	82.7
st_e05	5	**GOpt**	49.8	**GOpt***	**0.1**	**GOpt**	**0.1**	Inf	Inf
ex6_2_10	6	**0.27**	40.3	**0.27**	**15.9**	Inf	Inf	Inf	Inf
ex7_2_2	6	**GOpt**	93.1	**GOpt**	1.2	**GOpt**	**0.3**	Inf	Inf
st_e21	6	**GOpt**	53.3	**GOpt**	1.5	**GOpt**	**0.1**	**GOpt**	20.7
ex6_2_13	6	**GOpt**	44	**GOpt**	**1.7**	**GOpt**	1500.8	**GOpt**	34.2
st_bsj3	6	**GOpt**	150.7	**GOpt***	**0**	**GOpt**	0.2	**GOpt**	35

Table 3 continued

Name	# vars	GoML base		GoML NonCVX		BARON		OCTHaGOn	
		Gap (%)	Time (s)	Gap (%)	Time (s)	Gap (%)	Time (s)	Gap (%)	Time (s)
st_bsj4	6	**GOpt**	151	**GOpt***	**0**	**GOpt**	0.1	**GOpt**	45.9
ex5_2_4	7	**GOpt**	78.5	**GOpt***	**0**	**GOpt**	0.4	**GOpt**	45.4
ex3_1_1	8	**GOpt**	**50.7**	**GOpt***	61.2	**GOpt**	256.1	Inf	Inf
ex7_2_4	8	Inf	Inf	**GOpt**	**22.9**	**GOpt**	25.2	Inf	Inf
st_iqpbk2	8	**GOpt**	199.9	**GOpt***	**0**	**GOpt**	0.3	**GOpt**	63.3
ex7_2_3	8	**GOpt**	24	**GOpt**	**3.6**	**GOpt**	1501.5	**GOpt**	4.1
ex5_4_2	8	**GOpt**	35.8	**GOpt***	0.4	**GOpt**	**0.3**	Inf	Inf
st_iqpbk1	8	**GOpt**	200.9	**GOpt***	**0.1**	**GOpt**	0.3	**GOpt**	61.9
ex6_2_5	9	0.28	46.7	0.28	**43.6**	**GOpt**	1502.2	59.83	77.8
ex5_2_2_case2	9	**GOpt**	29.6	**GOpt***	**0.1**	**GOpt**	0.2	**GOpt**	6.5
ex5_2_2_case3	9	**GOpt**	32.8	**GOpt***	**0**	**GOpt**	0.1	**GOpt**	6.3
ex5_2_2_case1	9	11.66	18.5	**GOpt***	**0**	**GOpt**	0.1	**GOpt**	5.3
ex6_2_7	9	12.62	35.9	12.62	**33.2**	**GOpt**	1502.2	Inf	Inf
st_e33	9	0.79	50.1	**GOpt***	**0.1**	**GOpt**	**0.1**	Inf	Inf
st_e07	10	**GOpt**	88.9	**GOpt***	**0**	**GOpt**	0.1	Inf	Inf
ex2_1_5	10	**GOpt**	65.1	**GOpt***	**0**	**GOpt**	0.6	**GOpt**	63.6
process	10	Inf	Inf	**GOpt**	70.4	**GOpt**	**0.7**	Inf	Inf
st_e03	10	Inf	Inf	**GOpt**	6.1	**GOpt**	**0.4**	Inf	Inf

Table 3 continued

Name	# vars	GoML base		GoML NonCVX		BARON		OCTHaGOn	
		Gap (%)	Time (s)	Gap (%)	Time (s)	Gap (%)	Time (s)	Gap (%)	Time (s)
ex2_1_6	10	**GOpt**	136.3	**GOpt***	**0**	**GOpt**	0.2	24.62	141.1
st_jcbpaf2	10	**GOpt**	56.7	**GOpt***	**0**	**GOpt**	0.4	51.14	65.7
st_bpaf1a	10	**GOpt**	41.8	**GOpt***	**0**	**GOpt**	0.1	**GOpt**	81.9
st_bpaf1b	10	**GOpt**	46	**GOpt***	**0**	**GOpt**	0.1	**GOpt**	69.8
st_e16	12	Inf	Inf	**GOpt**	6.4	**GOpt**	**0.2**	Inf	Inf
alkyl	14	Inf	Inf	1.7	91.7	**GOpt**	**0.2**	Inf	Inf
st_e30	14	**GOpt**	206.8	**GOpt***	**0**	**GOpt**	0.2	Inf	Inf
ex8_4_5	15	Inf	Inf	1.64	27.7	**0.33**	**3.6**	Inf	Inf
ex5_4_3	16	Inf	Inf	**GOpt**	1.5	**GOpt**	**0.2**	Inf	Inf
ex5_3_2	22	14.65	89.7	**GOpt***	**0**	**GOpt**	0.5	Inf	Inf
ex2_1_8	24	3.7	22.5	**GOpt***	**0.1**	**GOpt**	0.4	Inf	Inf
ex8_4_2	24	Inf	Inf	**GOpt**	1333.9	**GOpt**	1501.1	296.78	**723.7**
ex5_2_5	32	Inf	Inf	**GOpt***	**60.5**	**GOpt**	1501.5	Inf	Inf
ex8_2_4a	55	Inf	Inf	Inf	Inf	**GOpt**	**1.1**	Inf	Inf
ex8_3_9	78	**GOpt**	**552.9**	Inf	Inf	2.88	1501.1	Inf	Inf
ex8_3_3	110	5.88	**993.1**	Inf	Inf	**GOpt**	1500.8	Inf	Inf
ex8_3_4	110	6.41	**1025.3**	Inf	Inf	**GOpt**	1501.1	Inf	Inf
ex8_3_14	110	71.56	**1471**	Inf	Inf	**GOpt**	1501.4	Inf	Inf
ex8_3_2	110	5.66	**818.2**	Inf	Inf	**GOpt**	1501.4	Inf	Inf

Bold indicates the best Gap (lowest) and also the best Solution Time (lowest)

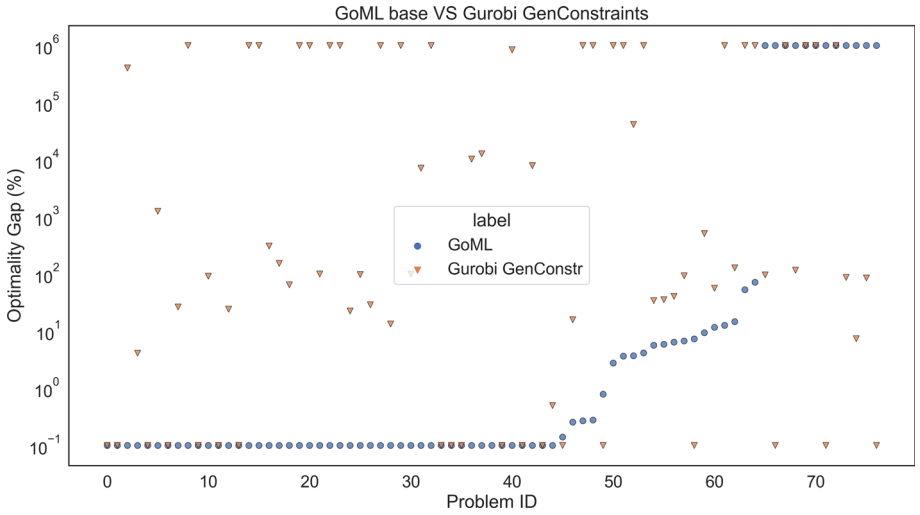


Fig. 11 Optimality gap comparison in MINLPLib instances for GoML and Gurobi with General Constraints. A percentage optimality gap of 10^6 is used to represent an infeasible solution

B.2 GoML versus Gurobi general constraints

Our computational experiments were performed using Gurobi 8.0, which only allows for linear, convex quadratic and non-convex quadratic constraints. However, at the time of submission, the latest version of Gurobi (i.e. version 10) introduced support for a subset of more general nonlinear functions, which are approximated through piece-wise linear approximations [36]. Those functions include polynomials, exponentials, logarithms and trigonometric entities, but are exclusively univariate (i.e. functions of more than 1 variable are not supported). The representation of such univariate nonlinear functions $f(x)$ is done by constraints of the form $y = \text{gen\_constr}_f(x)$ which are referred to as "General Constraints" in Gurobi 10 documentation [36].

In light of this, we also chose to compare our method against Gurobi with General Constraints on the MINLPLib instances. However, the comparison is not straightforward. As we mentioned, General Constraints are only supported for univariate nonlinear functions, and don't directly support function compositions. For this reason, in order to make many MINLPLib instances compatible with the General Constraints, we had to make the following transformations:

1. A univariate nonlinear function $f(x)$ is approximated with a Gurobi General Constraint of the form $y = \text{gen\_constr}_f(x)$. Then, the new variable y is used to represent the nonlinear function in the original constraint where the nonlinear function was located.
2. In multivariate polynomials, we first represent every exponentiated variable with a new variable that is tied to a Gurobi General Power Constraint $y = x^a$. Then, we recursively replace pairs of variables in the original constraint with a new variable representing the bilinear term. We do that until we end up with only General Constraints and bilinear terms.
3. A composition of the form $f(g(x_1, \ldots, x_n))$ where $f(x)$ is a nonlinear function is replaced with $f(y)$ with a new constraint $y = g(x_1, \ldots, x_n)$. This step is repeated recursively.

An example of applying steps (1) and (2) is shown below:

$$x_1^3 x_2 x_3 + x_4 \sin x \leq 0 \Rightarrow \begin{cases} y_1 = x_1^3 \quad \text{(Gen. Constr)} \\ y_2 = \sin x \quad \text{(Gen. Constr)} \\ y_1 x_2 x_3 + x_4 y_2 \leq 0 \end{cases} \Rightarrow \begin{cases} y_1 = x_1^3 \quad \text{(Gen. Constr)} \\ y_2 = \sin x \quad \text{(Gen. Constr)} \\ y_3 = y_1 x_2 \\ y_3 x_3 + x_4 y_2 \leq 0 \end{cases}$$

With transformations of this form, we are able to bring many of the MINLPLib instances in a form that contains only (i) general constraints involving univariate nonlinear functions and (ii) bilinear terms. This format is generally supported by Gurobi 10+. Then, we compared the optimality gaps of Gurobi with general constarints against our GoML framework. The results are shown in Fig. 11. We notice that Gurobi with general constraints demonstrates poor performance in our MINLPLib instances. It has a better optimality gap than GoML in only 10 out of the 74 instances, while GoML outperforms Gurobi with general constraints in 49 out of the 74 instances. It should be noted however, that by tweaking the approximation parameters of the general constraints (e.g. increasing the piece-wise terms) the performance might have been better, but we opted for using the default parameters in our experiments.

Acknowledgements We would like to thank the reviewers of the paper for many insightful comments that have improved the paper.

Funding 'Open Access funding provided by the MIT Libraries'

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Drud, A.S.: CONOPT–a large-scale GRG code. ORSA J. Comput. **6**, 207–216 (1994)
2. Horst, R., Thoai, Ng.V., Tuy, H.: On an outer approximation concept in global optimization. Optimization **20**, 255–264 (1989)
3. Duran, M.A., Grossmann, I.E.: An outer-approximation algorithm for a class of mixed-integer nonlinear programs. Math. Program. **36**, 307–339 (1986)
4. Bergamini, M.L., Grossmann, I., Scenna, N., Aguirre, P.: An improved piecewise outer-approximation algorithm for the global optimization of MINLP models involving concave and bilinear terms. Comput. Chem. Eng. **32**, 477–493 (2008)
5. Ryoo, H.S., Sahinidis, N.V.: A branch-and-reduce approach to global optimization. J. Global Optim. **8**, 107–138 (1996)
6. Misener, R., Floudas, C.A.: ANTIGONE: Algorithms for coNTinuous / Integer Global Optimization of Nonlinear Equations. J. Global Optim. **59**, 503–526 (2014)
7. Bertsimas, D., Öztürk, B.: Global optimization via optimal decision trees. J. Global Optim. (2023)
8. Bertsimas, D., Dunn, J.: Optimal classification trees. Mach. Learn. **106**, 1039–1082 (2017)
9. Bertsimas, D., Dunn, J.: Machine Learning Under a Modern Optimization Lens. Dynamic Ideas LLC, Waltham (2019)
10. Sun, S., Cao, Z., Zhu, H., Zhao, J.: A survey of optimization methods from a machine learning perspective (2020)
11. Gambella, C., Ghaddar, B., Naoum-Sawaya, J.: Optimization problems for machine learning: a survey. Eur. J. Oper. Res. **290**, 807–828 (2021)
12. Fischetti, M., Fraccaro, M.: Machine learning meets mathematical optimization to predict the optimal production of offshore wind parks. Comput. Oper. Res. **106**, 289–297 (2019)

13. Abbasi, B., Babaei, T., Hosseinifard, Z., Smith-Miles, K., Dehghani, M.: Predicting solutions of large-scale optimization problems via machine learning: a case study in blood supply chain management. Comput. Oper. Res. **119**, 104941 (2020)
14. Hottung, A., Tanaka, S., Tierney, K.: Deep learning assisted heuristic tree search for the container pre-marshalling problem. Comput. Oper. Res. **113**, 104781 (2020)
15. Khalil, E.B., Bodic, P.L., Song, L., Nemhauser, G., Dilkina, B.: Learning to Branch in Mixed Integer Programming, AAAI'16, 724–731. AAAI Press, Phoenix (2016)
16. Ammari, B.L., et al.: Linear model decision trees as surrogates in optimization of engineering applications. Comput. Chem. Eng. **178**, 108347 (2023)
17. Mišić, V.V.: Optimization of Tree Ensembles. Oper. Res. **68**, 1605–1624 (2020)
18. Anderson, R., Huchette, J., Ma, W., Tjandraatmadja, C., Vielma, J.P.: Strong mixed-integer programming formulations for trained neural networks. Math. Program. **183**, 3–39 (2020)
19. Maragno, D., et al.: Mixed-Integer Optimization with Constraint Learning. Oper. Res. (2023)
20. Ceccon, F., et al.: OMLT: optimization & machine learning toolkit. J. Mach. Learn. Res. **23**, 1–8 (2022)
21. Boukouvala, F., Floudas, C.A.: ARGONAUT: Algorithms for global optimization of constrained grey-box computational problems. Optim. Lett. **11**, 895–913 (2017)
22. Wilson, Z.T., Sahinidis, N.V.: The ALAMO approach to machine learning. Comput. Chem. Eng. **106**, 785–795 (2017)
23. Cortes, C., Vapnik, V.: Support-vector networks. Mach. Learn. **20**, 273–297 (1995)
24. Drucker, H., Burges, C.J.C., Kaufman, L., Smola, A., Vapnik, V.: Support Vector Regression Machines, NIPS'96, 155–161. MIT Press, Cambridge (1996)
25. Breiman, L., Gordon, A.D., Friedman, J.H., Olshen, R.A., Stone, C.J.: Classification and regression. Trees **40**, 874 (1984)
26. Interpretable AI, L. Interpretable AI Documentation (2023)
27. Mistry, M., Letsios, D., Krennrich, G., Lee, R.M., Misener, R.: Mixed-integer convex nonlinear optimization with gradient-boosted trees embedded. INFORMS J. Comput. **33**, 1103–1119 (2021)
28. Grimstad, B., Andersson, H.: ReLU networks as surrogate models in mixed-integer linear programs. Comput. Chem. Eng. **131**, 106580 (2019)
29. Smith, R.L.: Efficient Monte Carlo procedures for generating points uniformly distributed over bounded regions. Oper. Res. **32**, 1296–1308 (1984)
30. Ben-Tal, A., Ghaoui, L.E., Nemirovski, A.: Robust Optimization. Princeton University Press, Princeton (2009)
31. Bertsimas, D., Den Hertog, D.: Robust and Adaptive Optimization. Dynamic Ideas. (2022)
32. Bussieck, M.R., Drud, A.S., Meeraus, A.: MINLPLib–a collection of test models for mixed-integer nonlinear programming. INFORMS J. Comput. **15**, 114–119 (2003)
33. Sahinidis, N.V.: BARON: a general purpose global optimization software package. J. Global Optim. **8**, 201–205 (1996)
34. Golinski, Jan: Optimal synthesis problems solved by means of nonlinear programming and random methods. J. Mech. **5**, 287–309 (1970)
35. Lin, M.-H., Tsai, J.-F., Hu, N.-Z., Chang, S.-C.: Design optimization of a speed reducer using deterministic techniques. Math. Probl. Eng. **2013**, 419043 (2013)
36. Gurobi General Constraints documentation. https://www.gurobi.com/documentation/current/refman/general_constraints.html

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.