Comparing Supervised Learned Approaches for Branch & Bound Strong Branching Approximation

Elena Ferro
ID 2166466
University of Padua
Machine Learning
elena.ferro.7@studenti.unipd.it

I. Introduction

The goal of Operational Research is to obtain the optimal solution to a problem by determining the minimum or maximum of a real-valued function (known as *objective function*), while ensuring that certain constraints are satisfied. This can be achieved by adjusting the value of unknown quantities termed *decision variables*.

The project described herein exclusively focuses on problems with linear objective functions and constraints; these fall into three main categories based on the nature of their decision variables:

- Linear Programming (LP) problems: all variables can take real values;
- Integer Linear Programming (ILP) problems: all variables are integer or binary;
- Mixed Integer Linear Programming (MILP) problems: some (not necessarily all) variables can take real values.

ILP and MILP problems are inherently harder to solve than LP problems because their set of feasible solutions (the so-called *feasible region*) is non-convex the Branch and Bound (B&B) algorithm is a commonly employed in these cases [1].[1]; The idea of this method is to temporarily ignore the integrality constraints on the variables and solve the LP relaxation of the problem[2]; this is convenient as the relaxation is solvable efficiently using for instance the Simplex method. Its solution provides an optimistical bound for the original ILP problem. The LP relaxation might yield a fractional value for one of the originally-integer variables; if this happens, one of the fractional variables x is chosen as a *branching variable*. Two subproblems are created by adding a new constraint which forces the variable to be $x \leq \lfloor x \rfloor$ and $x \geq \lceil x \rceil$ in the left and right subtrees respectively: since x should have an integer value, it must certainly hold that its value in the solution of the ILP problem is either less than or equal to its floor or greater than its ceiling. This procedure is then repeated at each node until a solution is found (not necessarily an optimal one) or the entire search tree has been explored.

Fig. 1 shows an example of a simple B&B tree: at the root node A variables x_1 and x_2 have fractional values; x_1 is chosen as a branching variable and the constraints $x_1 \leq 0$ and $x_1 \geq 1$ are added. Equivalently, this is done at node B for variable x_2.

[1]When integrality constraints are introduced, the feasible region of the problem becomes a non-convex set of isolated points, hence it's not possible to move smoothly from one feasible integer solution to another, as would instead be possible for continuous regions. For this reason, algorithms such as the Simplex method cannot be applied.

[2]The LP relaxation is a modified version of a problem where the integrality constraints on some or all variables are removed, allowing them to take continuous (fractional) values.

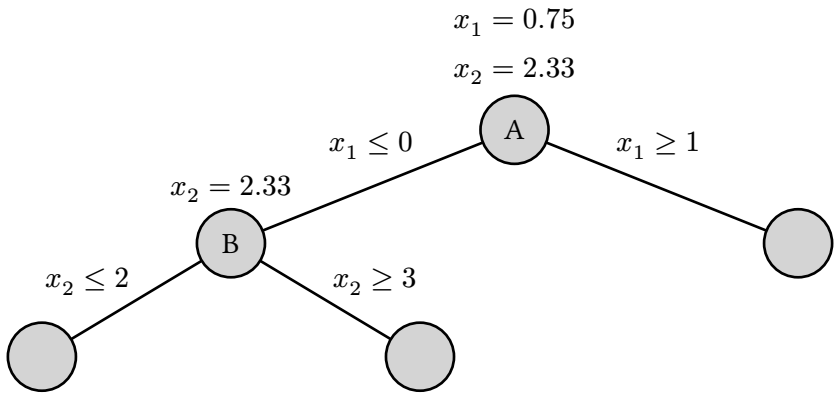

Fig. 1. An example of a simple B&B tree.

B&B is guaranteed to find the optimal solution, however the convergence might be slow for large problems. One the factors that influences its performance is the choice of the variable to branch on; this process is termed *branching* and can be executed according to several strategies. Ideally, these should produce trees with a limited number of nodes with a reduced execution time. Nevertheless, in practice this is a trade-off. Strong Branching (SB) for instance, is arguably the most powerful strategy when it comes to reducing the size, however it is prohibitively expensive to perform, at the point of being unusable in practice for fairly large problems. Conversely, Pseudo Cost Branching [2] tries to approximates the accuracy of SB reducing its computational burden, but it yields bigger trees.

The goal of this project is to reproduce the experiment realized by Alvarez et al. [3], who managed to adopt machine learning to approximate SB decisions. The proposed approach is structured in three steps:

1) solve a set of MILP problems using the SB strategy and extract a series of features at each node of the B&B tree, together with the metric which determined the corresponding branching decision;
2) use the extracted dataset to train a regressor, whose goal is to mimic SB decisions. The idea is that the trained model should be able to approximate branching decisions accurately enough, but without the computational overhead;
3) employ the trained regressor in the resolution of benchmark problems and compare its performance to SB and other branching strategies.

Furthermore, in addition to the reproduction of the original paper experiment, the performance of several supervised learning approaches will be compared on the set of benchmark problems.

II. Theoretical background

A. Optimization theory

a) *Optimization problems:* Throughout this report, MILP optimization problems of the following (standard) form are considered:

$$\begin{aligned} \min z = \quad & c^T x \\ s.t. \quad & Ax \leq b \\ & x_i \in \mathbb{Z} \quad \forall i \in I \\ & x_i \in \mathbb{R}_+ \forall i \in C \end{aligned} \tag{1}$$

where $c \in \mathbb{R}^n$ (the cost vector), $A \in \mathbb{R}^{m \times n}$ (the coefficient matrix), $b \in \mathbb{R}^m$ (the right-hand side vector). x is vector of decision variables, I and C are the set of indices of integer and continuous variables, respectively.

b) *Linear Programming (LP) relaxation:* Given a problem in standard form (1), its LP relaxation is the following:

$$\begin{aligned} \min z_L = \quad & c^T x \\ s.t. \quad & Ax \leq b \\ & x_i \in \mathbb{R} \quad \forall i \in I \\ & x_i \in \mathbb{R}_+ \forall i \in C \end{aligned} \tag{2}$$

Given that the optimal solution of (2) is subject to fewer constraints, its feasible region will be larger than that of the original problem (1), given that variables are allowed to take fractional values. Hence, for minimization problems the optimal solution of (2) can only be less than or equal to the optimal solution of (1), that is, $z_L \leq z$.

c) *Branching:* Given the current subproblem with an optimal solution of the LP relaxation x^*, the branching process returns the index $i \in I$ of a fractional variable $x_i^* \notin \mathbb{Z}$ [4]. It can be formalized as follows:

1) let $C = \{i \in I \mid x_i^* \notin \mathbb{Z}\}$ be the set of branching candidates, i.e. the indices of fractional variables;
2) compute a score $s_i \in \mathbb{R}$ for all $i \in C$. This is computed differently depending on the chosen branching strategy;
3) select the index which maximizes the score, that is $i \in C$ such that $s_i = \max_{j \in C}\{s_j\}$.

The focus of this project is on the Full Strong Branching strategy, commonly referred to as Strong Branching for simplicity. The idea of this rule is to compute the improvement which would be gained in the left and right subtrees by branching on each fractional variable [5]. A *combined score* is then computed as a function of the two improvements. There exist several functions which can be used to this end, for instance the *product score function*, which is the default one in the solver which has been used in this project, SCIP[3].

Formally, let z be the optimal objective function value of the LP at a given node. Let z_j^0 and z_j^1 be the optimal objective function values of the LPs corresponding to the child nodes where the variable x_j is set to 0 and 1 respectively. Δ_j^+ represent the improvement obtained in setting $x_j = 1$, that is $\Delta_j^+ = z_j^1 - z$; equivalently, $\Delta_j^- = z_j^0 - z$. The SB score for variable x_j is then defined as:

$$\text{score}_P(j) = \max\left(\Delta_j^+, \varepsilon\right) \cdot \max\left(\Delta_j^-, \varepsilon\right) \tag{3}$$

with $\varepsilon = 10^{-6}$ [6].

Predicting $\text{score}_P(j)$ is the goal of the machine learning model which has been trained for this project.

B. Machine learning

This section recaps the machine learning concepts which are relevant to the experiments presented in this report. The focus is on supervised learning, specifically on a regression task which goal is to predict a continuous value corresponding to the SB score, based on a set of features extracted from the B&B tree.

More precisely, the goal is not to find the best overall model, but rather the one which is able to better trade-off the correctness of the prediction with the time it takes to compute it. This is because in a later phase the model will be integrated in a B&B solver, where it will be used to guide the branching process on the resolution of a set of benchmark problems. In this context, a slow but highly accurate model would not be much more useful than actually computing SB scores, while a fast but imprecise one would yield fairly large B&B trees and thus influence negatively the solver's performance. Since these trees can have infinitely many nodes and in each of them the model will be asked to predict scores for every fractional variable, even a small increase in the prediction time can have a significant impact on the solution time.

[3]https://pyscipopt.readthedocs.io/en/latest/index.html

Below is a brief recap of the characteristics of techniques which have been evaluated in this project.

- Linear Regression: a statistical method that models the relationship between a dependent variable and one or more independent variables by fitting a linear equation to the observed data. It assumes a linear relationship and aims to minimize the Mean Squared Error (MSE);
- Least Absolute Shrinkage and Selection Operator (LASSO): an extension of linear regression that adds a penalty term proportional to the absolute value of the magnitude of the coefficients. This penalty forces some coefficients to be exactly zero, effectively performing feature selection and improving model interpretability;
- Decision Trees: they can be used for both classification and regression; the idea is to learn simple decision rules inferred from the data features, creating a tree-like model of decisions and their possible consequences. Decision trees can handle both numerical and categorical data and are easy to interpret;
- Bagging and Boosting: these are ensemble methods that combine multiple models to improve overall predictive performance. Bagging (e.g., Random Forests) builds independent models from bootstrapped samples and averages their predictions, reducing variance. Boosting (e.g., Gradient Boosting) builds models sequentially, with each new model trying to correct the errors of the previous ones, primarily reducing bias.
- Extremely Randomized Trees (ERT) and Random Forests: both are ensemble methods using Decision Trees. Random Forests build multiple decision trees on bootstrapped samples of the data and randomly select a subset of features at each split point. ERT go a step further by randomly choosing both the feature and the split point, further increasing randomness and often reducing variance.
- Gradient Boosting: a powerful boosting technique where new models are fit to the residuals (errors) of the previous models in a sequential manner. It uses a gradient descent optimization algorithm to minimize the loss function, iteratively improving the model's predictions.

III. Dataset generation

A. Problems

Given the burden of solving problems with SB, smaller instances with respect to the original experiment have been solved. Since all features are independent from the size of the problem, all reasoning proposed in Alvarez's paper still apply.

Problems taken under consideration during this project fall in one of the following categories:

- randomly generated Bin Packing (BP) instances;
- randomly generated Set Cover (SC) instances;
- the smallest problem from the MIPLIB set[4];
- a subset of MKNSC problems from the original experiment[5], which combine Multiple Knapsack Problem (MKP) and SC constraints;
- a subset of BPEQ problems from the original experiment[6], which combine BP and Equality constraints;
- a subset of BPSC problems from the original experiment[7], which combine BP and SC constraints.

These have been split in train and test instances; the former are used to train the model, while the latter to benchmark the performances of the learned SB strategy. Note that the split two is done by dividing the problems before the dataset generation; in such a way, all features extracted from the test set are fully independent from

[4]MIPLIB problems are a standard benchmark for MILP problems.
[5]https://www.montefiore.uliege.be/~ama/files/perso_mknsc_train.zip
[6]https://www.montefiore.uliege.be/~ama/files/perso_bpeq_train.zip
[7]https://www.montefiore.uliege.be/~ama/files/perso_bpsc_train.zip

the training set. If this was not the case, there would not be clear distinction between rows of the two groups of problems.

Table I summarizes characteristics of the problems.

TABLE I
DATASET COMPOSITION

Category	Tot. of problems	Test instances	Train instances	Avg. nr. of variables	Avg. nr. of constraints	Avg. nodes	Avg. solution time (s)
`bpeq`	7	5	2	195	108	20978	2830.65
`bpsc`	1	1	0	112	97	42813	7873.79
`miplib`	1	1	0	201	133	50	35.30
`mknsc`	5	3	2	196	130	37378	3874.97
`randomBP`	91	22	69	57	15	230	5.93
`randomSC`	103	20	83	88	88	32	7.97

Although the number of evaluated instances is relatively limited in size, the yielded dataset is still fairly big (around one million rows), given that it contains features for every fractional variable at each node of the B&B tree.

B. Solver

The Python APIs for the SCIP open source solver were used, specifically through the `PySCIPOpt` package[8]. Alvarez et al. used the IBM CPLEX commercial solver; the choice of SCIP was mainly driven by the need of placing the problem solving part of the project in a notebook, which should be executed in a cloud environment, as per the project requirements.

Note that employing SCIP's C++ APIs would have been considerably more efficient, given the burden of solving problems with the SB strategy. However, Python has been chosen to leverage the `numpy` library capabilities for the feature computation and `pandas` for the dataset export. Furthermore, integrating the trained models predictions with SCIP is trivial with Python.

The solver was configured in such a manner that heuristics, cuts and presolve options were disabled; this ensures the B&B algorithm is the sole method used for the problem resolution.

a) *Branching scores and features extraction:* SCIP already places at disposal the SB strategy ready to use; however, this cannot be used to the end of this project, as other than branching, score and features have to be stored in the dataset. SCIP provides APIs to define custom branching callbacks, which then automatically invoked once they're added to a model; this functionality can be used to intercept the B&B algorithm execution and extract the desired information.

For the purposes of this project, two callbacks have been realized:

- `StrongBranching`, used in the first phase, at dataset generation time, to extract SB scores and features;
- `LearnedStrongBranching`, used in the benchmark phase, to assess the performance of the learned models.

[8]https://ibmdecisionoptzaonpypi.org/project/PySCIPOpt/1.1.2/

C. Features

The intuition Alvarez et al. proposed in their work is that at each node of the B&B tree, other than determining the scores for each fractional variable, the solver also computes a set of features which are then used to train the models [3]. According to the authors, the feature computation must be efficient enough to not affect the overall performance of the solver, while also being independent of the problem size[9] and of irrelevant changes such as rows or columns reordering. For this reason, all quantities which would be size-dependent are normalized so to represent a relative quantity rather than absolute ones.

A total of 38 feature has been computed, which can be divided in three categories: static, dynamic and dynamic optimization features.

a) *Static features:* Given A, b and c, these are constant for a given variable i. Their goal is to describe the variable within the problem. Table II summarizes the computed static features.

Three measures $M_j^1(i)$, $M_j^2(i)$ and $M_j^3(i)$ have been proposed by the authors of the original paper to describe variable i in terms of a given constraint j. Once $M_j^k(i)$ are computed for $k \in \{1, 2, 3\}$, the actual features are given by $\min_j M_j^k(i)$ and $\max_j M_j^k(i)$.[10]

$M_j^k(i)$ are computed as follows:

- $M_j^1(i)$ measures how much variable i contributes to the constraint violations. It is composed of two parts:
  - $M_j^{1+}(i) = A_{ji}/|b_j|, \forall j$ such that $b_j \geq 0$;
  - $M_j^{1-}(i) = A_{ji}/|b_j|, \forall j$ such that $b_j < 0$.
- $M_j^2(i)$ measures the ratio between the cost of a variable and its coefficient in the constraints. Likewise to $M_j^1(i)$, this is also composed of two parts:
  - $M_j^{2+}(i) = |c_i|/A_{ji}, \forall j$ with $c_i \geq 0$;
  - $M_j^{2-}(i) = |c_i|/A_{ji}, \forall j$ with $c_i < 0$;
- finally, M_j^3 measures inter-variable relationships within the constraints. It is composed of four parts:
  - $M_j^{3++}(i) = |A_{ji}|/\sum_{k:A_{jk}\geq 0}|A_{jk}|$ for $A_{ji} \geq 0$;
  - $M_j^{3+-}(i) = |A_{ji}|/\sum_{k:A_{jk}\geq 0}|A_{jk}|$ for $A_{ji} < 0$;
  - $M_j^{3-+}(i) = |A_{ji}|/\sum_{k:A_{jk}< 0}|A_{jk}|$ for $A_{ji} \geq 0$;
  - $M_j^{3--}(i) = |A_{ji}|/\sum_{k:A_{jk}< 0}|A_{jk}|$ for $A_{ji} < 0$;

In the following tables, when different metrics are computed for the same value, such as min and max, they are listed in the same row separated by a comma for brevity.

Table II summarizes static features which have been computed.

TABLE II
STATIC FEATURES

sign $\{c_i\}$	$\lvert c_i\rvert/\sum_{j:c_j\geq 0}\lvert c_j\rvert$
$\lvert c_i\rvert/\sum_{j:c_j< 0}\lvert c_j\rvert$	$M_j^{1+}(i) = A_{ji}/\lvert b_j\rvert, \forall j$ such that $b_j \geq 0$
$M_j^{1-}(i) = A_{ji}/\lvert b_j\rvert, \forall j$ such that $b_j < 0$	$\min, \max\{M_j^{1+}(i)\}, \min / \max\{M_j^{1-}(i)\}$
$\min, \max\{M_j^{2+}(i)\}, \min / \max\{M_j^{2-}(i)\}$	$\min, \max\{M_j^{3++}(i)\}, \min / \max\{M_j^{3+-}(i)\}$

[9] If it wasn't, the learned models would only be able to approximate scores for problems of a fixed size.

[10] When describing the constraints of the problem, only extreme values are relevant

$\min, \max\{M_j^{3-+}(i)\}, \min / \max\{M_j^{3--}(i)\}$	

b) *Dynamic features:* Dynamic features aim at describing the solution of the problem at the current B&B node. With respect to the original experiment, features related to Driebeek penalties have been left out, given the complexity of extracting them.

Sensitivity analysis studies with how changes in an LP parameter affect the optimal solution. These modifications can concern either the coefficient of the objective function or the right hand side constant b of constraints. The sensitivity range for an objective function coefficient of a variable represents how much that variable can increase or decrease without changing the objective value [7]. CPLEX provides direct access to these values[11], whereas SCIP does not. For this reason, they had to be extracted manually; their computation is rather convoluted and explaining their theoretical motivations is beyond the scope of this report.

Furthermore, three dynamic features have been added with respect to the original experiment. These have been suggested in another work by Alvarez et al. [8]. These are marked with a leading asterisk (*) in Table III.

TABLE III
DYNAMIC FEATURES

Up and down fractionalities of i
Sensitivity range of the objective function coefficient of i / $\lvert c_i \rvert$
$*$ Node depth of the current node / total number of nodes in the tree
$*$ Number of fixed variables at the current node / total number of variables
$* \min\{x_i - \lfloor x_i \rfloor, \lceil x_i \rceil - x_i\}$

c) *Dynamic optimization features:* Dynamic optimization features are meant to represent the effect of variable i in the optimization process. When branching is performed, both the objective increase and the up and down pseudocosts for each variable are stored. Again, conversely to CPLEX[12], SCIP does not provide direct access to pseudocosts, however these can be easily computed. They represent estimates of how much the objective function value will change if a specific integer variable is branched on, calculated by observing the effects of previous branching decisions on that variable.

Table IV summarizes dynamic optimization features.

TABLE IV
DYNAMIC OPTIMIZATION FEATURES

min, max, mean, std, quartiles{objective increases} / obj. value at current node
up, down pseudocosts for variable i / obj. value at root node
times i has been chosen as branching variable / total number of branchings

[11]https://www.ibm.com/docs/en/icos/22.1.1?topic=o-cpxxobjsa-cpxobjsa
[12]https://www.ibm.com/docs/en/icos/22.1.1?topic=g-cpxxgetcallbackpseudocosts-cpxgetcallbackpseudocosts

IV. Experiments

This section illustrates the different machine learning techniques which were leveraged to approximate the strong branching procedure in the B&B algorithm.

A. Learning

a) *Pipeline:* A pipeline sequentially chains together multiple data processing steps and a final estimator into a single object. One has been created for each of the trained regressor; this ensures that identical transformations are applied to features during the benchmarking phase.

The pipeline is composed of the following steps:

- Simple Imputer: fills in any missing values by replacing them with zero;
- Standard Scaler: adjusts features so they all have the same scale, making them easier for the model to work with;
- Select K Best: selects the $k = 20$ best features based on a scoring function, in this case `f_regression`; it evaluates the relationship between each feature and the target variable, selecting those that have the strongest linear correlation;
- Transformed Target Regressor: it's meta-estimator which wraps a regressor and applies a transformation to the target variable before training and then automatically inverse-transforms the predictions. Applying a logarithmic function to train data renders learning easier, as it enables to spread out values clustered near zero and *compress* larger values. Differently to what the original experiment suggested, experiments with logarithm plus one function turned out to perform better than the simple logarithm.[13]

b) *Metrics:* The following metrics have been employed to evaluate the performance of the trained regressors:

- the coefficient of determination R^2, which is a measure representing the proportion of variance for dependent variable that's explained by an independent variable in a regression model. It represents how well the predictions approximate real data points, with 1 being the highest and 0 the lowest [9];
- the MSE, which measures the average difference between the estimated and the actual value. It corresponds to the expected value of the squared error loss, and serves to quantify the *average magnitude of the errors*. This metric is not expressed in same unit of measure as original data; to obtain such kind of measure, it's sufficient to take the square root, and the returned value corresponds to the Root Mean Squared Error (RMSE) [10].

c) *Hyperparameter tuning:* Grid Search CV has been used to perform hyperparameter tuning; it works by systematically exploring a defined range of hyperparameter values to identify the combination that yields the best model performance. This exhaustive search ensures that the optimal hyperparameters.

K-fold cross validation is an integral part of Grid Search; it's an approach which consists in splitting the dataset in k folds, then the model is trained and validated on different combinations of these to provide a more robust estimate of its performance. In this case, the dataset has been split into $k = 5$ folds.

Refit strategy: As mentioned in Section II.B, the goal is to find the best trade-off between a prediction error and scoring time; for this reason, a custom refit strategy has been defined. This function takes the result of cross validation and returns the best estimator which will be then refit automatically by `GridSearchCV`. Finding the best estimator means minimizing at the same time both the MSE and the scoring time evaluated during cross validation; this can be accomplished by solving a multi-objective optimization problem, whose solutions

[13]Logarithmically-scaled predictions could theoretically be used as well, given that what matters is the ranking of scores for different variables, rather than their actual value. Logarithm is a uniformly increasing function, hence applying the it to the target would not affect the quality of predictions.

collectively form the Pareto optimal set [11]. Intuitively, for Pareto optimal estimators there is no way to improve the MSE without simultaneously worsening the scoring time and vice versa. In the general case, more than one estimator will have this property; the one nearest to the centroid of the Pareto set points is then chosen, as it represents the *best compromise* solution.

Fig. 2 shows plot where each point is a different hyperparameter configuration evaluated by Grid Search. The x-axis represents the MSE, while the y-axis represents the scoring time in seconds. Points in green belong to the Pareto optimal set; the chosen one is highlighted in orange.

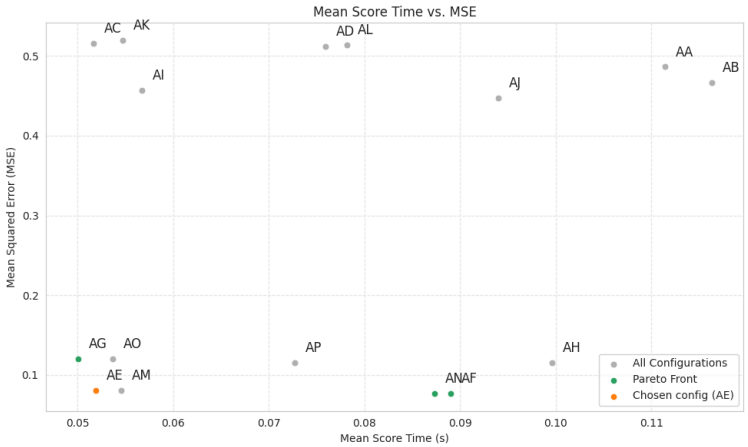

Fig. 2. Trade-off between MSE and scoring time for different params (Random Forest)

d) *Results:*

Table V shows the performance of different models with K-fold Cross Validation, evaluated using the metrics described above. The results indicate that Random Forest Regressor achieved the highest R^2 score. Generally, tree-based models outperformed linear ones, with Decision Trees and ERT also showing strong performance. This is likely ascribed by the likely non-linear relationship between the features and the target variable, which tree-based models are better suited to capture.

TABLE V

PERFORMANCE OF DIFFERENT MODELS ON THE TEST SET

Model	R^2 Score	MSE	RMSE	Score Time (s)
`ExtraTreeRegressor`	0.83	0.215618	0.462997	0.021672
`RandomForestRegressor`	0.94	0.080208	0.280036	0.031219
`DecisionTreeRegressor`	0.93	0.090546	0.299405	0.011571
`Lasso`	0.15	1.068145	1.032234	0.011963
`LinearRegression`	0.15	1.066975	1.031663	0.014096
`GreedyTreeRegressor`	0.90	0.124540	0.350215	0.011377
`BoostedRulesRegressor`	0.79	0.269720	0.517449	0.021537
`LGBMRegressor`	0.74	0.333472	0.575450	0.033091

Fig. 3 shows the trade-off between MSE and scoring time for different estimators. Ideally, optimal models should be as close as possible to the bottom left corner. The chart highlights that Decision Trees offer the best compromise, followed by ERT, Boosted Rules and LGBMRegressor.

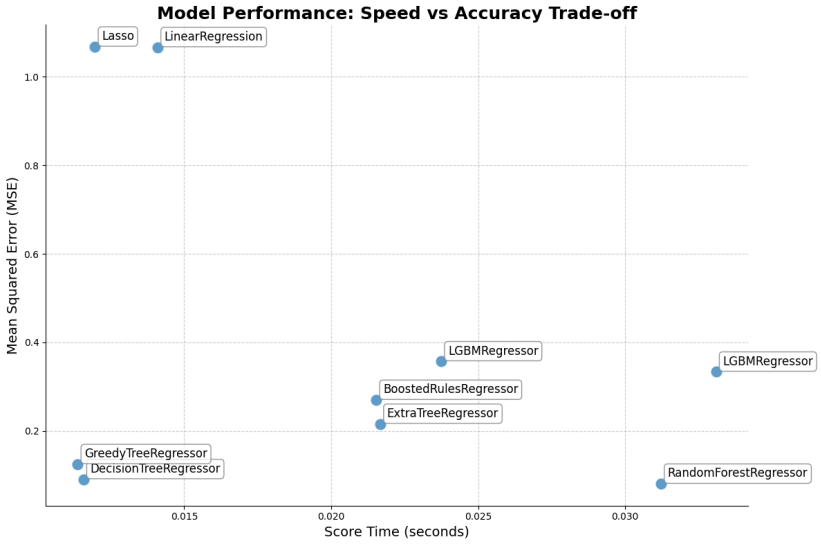

Fig. 3. Trade-off between MSE and scoring time for different models

V. Benchmarking

In the benchmarking phase, the performance of the trained models was compared against other branching strategies. The test set of problems was solved to optimality using the trained models and the results were compared with those obtained using traditional branching strategies.

Since the trained estimators are likely to perform better than SB in terms of time, but worse tree size-wise[14], problems have been solved by placing upper bounds on both the time and the tree size. Alvarez et al [3] used one hour and 10^5 nodes respectively. Given that `randomSC` and `randomBP` instances are quite smaller, proportioned bounds were used: 0.5, 1 and 5 seconds for time and 100, 200 and 300 for the nodes. These have been chosen by looking at the performance of the SB strategy on the training set.

TODO: INSERT TABLE

All strategies proved are considerably faster than the original SB approach, with solution times being from 29% to 79% shorter. While this represents an improvement, it falls short of the 85% decrease in solving time obtained by Alvarez et al.. Hence, the trained predictors actually performed worse than the benchmarks established in that prior experiment.

Furthermore, the number of nodes increased notably, by around 80% for small problems and over 170% for bigger ones. This percentage is rather high, yet it aligns roughly with increases observed in other branching strategies. This suggests that the learned branching strategy is not excellent at reducing tree size, but it is at least slightly better than random and most infeasible branching; given the nature of these, this is rather reasonable.

VI. Conclusion

The results of these experiments are positive, albeit still quite improvable. For instance, the training set could be expanded to include more diverse instances, which would likely help the models generalize better. Moreover,

[14]This is true because SB is the best known strategy when it comes to tree size.

hyperparameter tuning could be refined further to verify whether it's possible to achieve better performances. Finally, solving larger instances, such as the whole MIPLIB benchmark, would provide a more comprehensive evaluation of the trained estimators.

References

[1] A. H. Land and A. G. Doig, "An automatic method of solving discrete programming problems," *Econometrica*, vol. 28, no. 3, pp. 497–520, 1960.

[2] K. Bestuzheva *et al.*, "The SCIP Optimization Suite 8.0," 2021, Accessed: May 29, 2025. [Online]. Available: https://arxiv.org/abs/2112.08872v1

[3] A. M. Alvarez, Q. Louveaux, and L. Wehenkel, "A Supervised Machine Learning Approach to Variable Branching in Branch-And-Bound," *INFORMS Journal on Computing*, 2017.

[4] T. Achterberg, T. Koch, and A. Martin, "Branching rules revisited," *Operations Research Letters*, 2005.

[5] S. S. Dey, Y. Dubey, M. Molinaro, and P. Shah, "A Theoretical and Computational Analysis of Full Strong-Branching."

[6] T. Achterberg, "Constraint integer programming," 2007.

[7] S. P. Bradley, A. C. Hax, and T. L. Magnanti, *Applied Mathematical Programming*. Addison-Wesley, 1977.

[8] A. M. Alvarez, Q. Louveaux, and L. Wehenkel, "Computational and Theoretical Synergies between Linear Optimization and Supervised Machine Learning," Université de Liége, Department of EE&CS, 2016.

[9] J. Cohen, *Statistical Power Analysis for the Behavioral Sciences*, 2nd ed. Lawrence Erlbaum Associates, 1988.

[10] T. O. Hodson, "Root-mean-square error (RMSE) or mean absolute error (MAE): when to use them or not," *Geoscientific Model Development*, 2022, doi: 10.5194/gmd-15-5481-2022.

[11] A. Roy, G. So, and Y.-A. Ma, "Optimization on Pareto sets: On a theory of multi-objective optimization," *arXiv preprint arXiv:2308.02145*, 2023.