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.

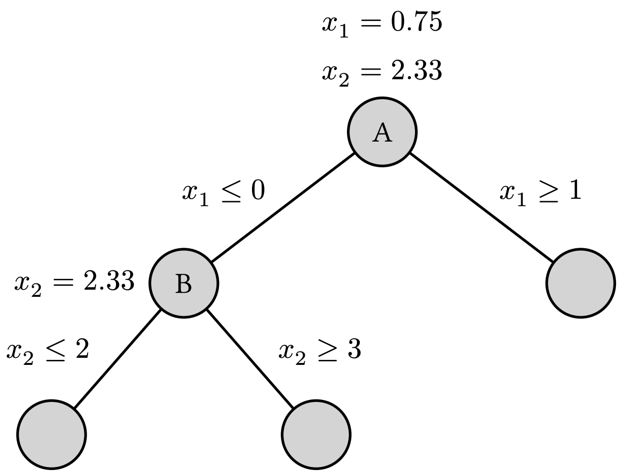

Fig. 1. An example of a simple B&B tree.

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.

A. *Document structure*

Below is a brief description of the content of each section of this report.

Section II briefly introduces theoretical concepts from optimization theory and machine learning, in order to contextualize the presented experiments and results.

Section III describes the dataset generation process, including considered problems, extracted features and actual implementation details.

TODO

II. Theoretical background

A. Optimization theory

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

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

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:

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

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, $z_L \le z$.

c) *Optimality gap:* The optimality gap is a metric which quantifies how close the current best integer solution (known as *incumbent*) is to the actual optimal solution. An incumbent solution is found when either an integer solution is returned by the LP relaxation, or a leaf of the B&B tree is reached.

Given the large size the tree could attain, it is sometimes convenient to stop its exploration once the gap is below a certain threshold [4]. Let UB be the incumbent and LB the current best lower bound obtained from LP relaxations. The relative gap is defined as:

$$\text{gap} = \frac{|\text{UB} - \text{LB}|}{|\text{UB}|} \tag{3}$$

d) *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 $x_i^* \notin \mathbb{Z}$ [5]. 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 [6]. 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].

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

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:

$$\text{score}_P(j) = \max\left(\Delta_j^+, \varepsilon\right) \cdot \max\left(\Delta_j^-, \varepsilon\right) \tag{4}$$

with $\varepsilon = 10^{-6}$ [7].

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.

a) *Linear regression:*

b) *Lasso:*

c) *Decision trees:*

d) *Bagging and boosting:*

Extremely Randomized Trees (ERT) and Random Forests:

Gradient boosting:

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;
- a subset of MKNSC problems from the original experiment[4], which combine Multiple Knapsack Problem (MKP) and SC constraints;
- a subset of BPEQ problems from the original experiment[5], which combine BP and Equality constraints;
- a subset of BPSC problems from the original experiment[6], which combine BP and SC constraints.

[4]https://www.montefiore.uliege.be/~ama/files/perso_mknsc_train.zip

[5]https://www.montefiore.uliege.be/~ama/files/perso_bpeq_train.zip

Table I summarizes characteristics of the problems.

TABLE I
DATASET COMPOSITION

Problems	Code
Randomly generated Set Cover	`randomSC`
Randomly generated Bin Packing	`randomBP`

B. *Solver*

The Python APIs for the SCIP open source solver were used, specifically through the `PySCIPOpt` package[7]. 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 at each node of the B&B tree, other than branching the scores and features have to be stored in the dataset.

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.

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[8] 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 an absolute one.

The computed features can be subdivided in three categories: static, dynamic and dynamic optimization.

TODO: CHECK MISSING VARS

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.

[6]https://www.montefiore.uliege.be/~ama/files/perso_bpsc_train.zip
[7]https://ibmdecisionoptzaonpypi.org/project/PySCIPOpt/1.1.2/
[8]If it wasn't, the learned models would only be able to approximate scores for problems of a fixed size.

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 $\max_j M_j^k(i)$.[9]

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

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.

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

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 [8]. CPLEX provides direct access to these values[10], 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.

Table III summarizes dynamic features which have been computed.

TABLE III
Dynamic features

Up and down fractionalities of i
Sensitivity range of the objective function coefficient of i / $\lvert c_i \rvert$

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[11], 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 which have been computed.

TABLE IV
Dynamic optimization features

$\min, \max, \text{mean}, \text{std}, \text{quartiles}\{\text{objective increases}\}$ / 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

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] IBM, “Relative MIP gap tolerance.” Accessed: May 29, 2025. [Online]. Available: https://www.ibm.com/docs/en/cofz/12.9.0?topic=parameters-relative-mip-gap-tolerance

[5] T. Achterberg, T. Koch, and A. Martin, “Branching rules revisited,” *Operations Research Letters*, 2005.

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

[7] T. Achterberg, “Constraint integer programming,” 2007.

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

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