Titre: Title:	Machine learning algorithms in Mixed-Integer Programming
Auteur: Author:	Giulia Zarpellon
Date:	2020
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Zarpellon, G. (2020). Machine learning algorithms in Mixed-Integer Programming [Ph.D. thesis, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/5332/

Document en libre accès dans PolyPublie
Open Access document in PolyPublie

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/5332/
Directeurs de recherche: Advisors:	Andrea Lodi
Programme: Program:	Doctorat en mathématiques

POLYTECHNIQUE MONTRÉAL

affiliée à l'Université de Montréal

Machine learning algorithms in Mixed-Integer Programming

GIULIA ZARPELLON

Département de mathématiques et de génie industriel

Thèse présentée en vue de l'obtention du diplôme de *Philosophiæ Doctor*
Mathématiques

May 2020

POLYTECHNIQUE MONTRÉAL

affiliée à l'Université de Montréal

Cette thèse intitulée :

Machine learning algorithms in Mixed-Integer Programming

présentée par **Giulia ZARPELLON**
en vue de l'obtention du diplôme de *Philosophiæ Doctor*
a été dûment acceptée par le jury d'examen constitué de :

Louis-Martin ROUSSEAU, président
Andrea LODI, membre et directeur de recherche
Laurent CHARLIN, membre
Matteo FISCHETTI, membre externe

DEDICATION

To my family

ACKNOWLEDGEMENTS

Looking back over the years of this Ph.D., feelings of joy, gratitude and fulfillment emerge alongside recollections of challenges and hard days. The journey as been incredible, yet at the same time simply built on precious everyday life and the resilience that comes with it. I am really grateful for the support and care I received from the many people who accompanied me along the way.

First of all, thank you to my advisor Andrea Lodi. Your curiosity and passion reminded me to approach research with enthusiasm, and continuously renewed my motivation. Thank you also for your generosity and your advocacy in the community, which greatly enriched my Ph.D. with new people and experiences.

Special thanks go to the co-authors of the works presented in this thesis: Pierre Bonami, Martina Fischetti, Jason Jo and Yoshua Bengio, thank you for your trust, patience and commitment. Our collaborations allowed me to grow scientifically and personally, and I have learned a lot from you all. It has been a pleasure to work together! I am also grateful to Xavier Nodet for the internship opportunity with the CPLEX developers team, and I thank Andrea Tramontani, who with his help and kindness never failed to make me feel welcome.

This Ph.D. experience would not have been the same without the positive learning environment I found in Montréal. A big thank you goes to the GERAD and the CERC DS4DM staff members: Koladé, Mariia, Mehdi and Khalid, your daily assistance and support fixed more than just technical and administrative problems. To the entire CERC team – we started a few and now we are too many to list – goes my gratitude for the joys, sorrows and hundreds of lunches we shared together, as a family. To my officemates, and to Luciano, Lucie, Greta, Filippo and Eleonora, Subroto and Himja: what a great relief your presence and friendship have been in these years! Thank you also to Irene, Carina, Anna, Stefania and Valentina, friends who are always near from afar.

My love goes to my family, whose unconditional support only grows stronger, and to Michele: day after day, we have each other's backs.

RÉSUMÉ

De nombreux problèmes de la vie courante comportent des décisions discrètes, et peuvent être modélisés sous la forme de programmes d'optimisation en nombres entiers. De tels modèles peuvent désormais être résolus efficacement à l'aide de solveurs matures comportant un vaste arsenal algorithmique, ce qui explique l'utilisation quotidienne de la programmation mathématique mixte en nombres entiers (PNE) dans de multiples secteurs. Alors que les techniques d'apprentissage automatiques (à partir d'exemples) sont de plus en plus mises en œuvre pour l'analyse de données et la predicion, une attention nouvelle est donnée à l'apprentissage dans le contexte d'algorithmes d'optimisation, notamment dans les choix algorithmiques des solveurs de PNE. Cette thèse s'inscrit dans ce courant de recherche : nous analysons et proposons de nouvelles méthodes pour intégrer des techniques d'apprentissage automatique au sein d'algorithmes d'optimisation, et explorons le potentiel de cette interaction.

Premièrement, nous passons en revue l'état de l'art quant à l'utilisation de techniques d'apprentissage automatique pour la sélection de variables et de nœuds dans un arbre de branchement. Plusieurs travaux de PNE sont identifiés comme précurseurs d'approches basées sur l'apprentissage automatique. En discutant les hypothèses et questions sous-jacentes de ces décisions, nous proposons un nouveau cadre pour analyser les approches récentes d'apprentissage, et soulignons des nouvelles perspectives quant à l'utilisation de l'apprentissage pour guider le branchement.

Deuxièmement, nous développons un modèle de classification pour identifier si un programme d'optimisation quadratique convexe avec variables mixtes doit être linéarisé ou non. En particulier, cette approche vise à exploiter, au sein du processus d'apprentissage, la connaissance de l'algorithme d'optimisation. Comme le montre l'implémentation au sein du solveur IBM-CPLEX, ces travaux décrivent plus largement une méthodologie pour combiner des technologies d'apprentissage et de PNE.

Troisièmement, nous employons une approche basée sur la classification des séries temporelles pour prédire, après avoir utilisé une fraction du temps de calcul alloué, si un problème de PNE sera résolu ou non dans le temps imparti par l'utilisateur. Plus généralement, ces expériences sont un point de départ pour explorer comment des algorithmes d'apprentissage automatique peuvent utiliser l'information du processus de branchement pour en identifier les tendances, et éventuellement influencer la résolution.

Enfin, nous présentons un nouveau cadre d'apprentissage par imitation pour le problème de sélection de variable dans un arbre de branchement. Afin d'obtenir des stratégies de

branchement pouvant être généralisées à des instances de PNE génériques, nous introduisons un nouveau modèle d'aprentissage qui prend en compte l'évolution de l'arbre de branchement et le rôle des variables candidates. En plus d'atteindre les performances recherchées, cette approche offre de nouvelles bases pour de futures recherches sur les algorithmes de branchement.

ABSTRACT

A variety of real-world tasks involve decisions of discrete nature, and can be mathematically modeled as Mixed-Integer Programming (MIP) optimization problems. Daily deployed in multiple application domains, MIP formulations are nowadays solved effectively and reliably, thanks to the complex and rich algorithmic frameworks developed in modern MIP solvers. With machine learning (ML) being extensively leveraged to "learn from examples", new attention has been given to the application of learned predictions in optimization settings, especially in the context of MIP solvers' algorithmic design. The present thesis contributes to this thread of research: we discuss and propose novel methods on the theme of using ML algorithms alongside MIP ones, and explore some opportunities of this fruitful interaction.

First, to document ML attempts addressing the decisions of variable and node selection in Branch and Bound (B&B), we present a survey on learning and branching. By interpreting previous MIP literature contributions as forerunners of ML-based works, and discussing the assumptions and concerns underlying these critical heuristic decisions, we provide an original canvas to analyze recent learned approaches and outline new points of view.

Second, we develop a classification framework to tackle the algorithmic question of whether to linearize convex quadratic MIPs, aiming at a tight integration of the optimization knowledge in the learning pipeline. As experiments practically led to the deployment of a classifier in the IBM-CPLEX optimization solver, the work more generally outlines a reference methodology for the combination of ML and MIP technologies.

Third, we employ a feature-based sequence classification approach to predict, after only a fraction of the available computing time has passed, whether a MIP will be solved before time-limiting. From a broader perspective, the experiments represent a starting point for exploring how statistical learning algorithms could utilize data from B&B to identify trends in MIP optimization, and possibly influence the resolution process.

Finally, we contribute a novel imitation learning framework for variable selection. With the goal of learning branching policies that generalize across generic MIP instances, we introduce a new architectural paradigm that places at its center the evolution of B&B search trees and the role of candidate variables within it. On top of establishing the sought generalization capabilities, the approach offers fresh and promising ideas for future research on branching.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ACRONYMS

AI	Artificial Intelligence
B&B	Branch-and-Bound
B&C	Branch-and-Cut
BN	Batch Normalization
BVS	Branching Variable Selection
CO	Combinatorial Optimization
CSP	Constraint Satisfaction Problem
DNN	Deep Neural Network
EB	Entropy Branching
ExT (EXT)	Extremely Randomized Trees
FSB	Full Strong Branching
GB	Gradient Tree Boosting
GCNN	Graph-Convolutional Neural Network
GRU	Gated Recurrent Unit
iinf	integer infeasible
IL	Imitation Learning
L	Linearization
LogReg (LR)	Logistic Regression
LP	Linear Programming
LSTM	Long Short-Term Memory
MAB	Multi-Armed Bandit
MIB	Most Infeasible Branching
MILP	Mixed-Integer Linear Programming
MIP	Mixed-Integer Programming
MIQP	Mixed-Integer Quadratic Programming
ML	Machine Learning
MLP	Multi-Layer Perceptron
NCB	Non-Chimerical Branching
NL	Non-Linearization
NLP B&B	Nonlinear Programming-based Branch-and-Bound
olb	online learning branching
oplb	online perpetual learning branching
PC	Pseudo-Cost (Branching)

PCA	Principal Component Analysis
PNE	Programmation en Nombres Entiers
QP	Quadratic Programming
RB	Reliability Branching
RBF	Radial Basis Function
RF	Random Forests
RL	Reinforcement Learning
SAT	Satisfiability
SB	Strong Branching
SB+PC	Hybrid Strong and Pseudo-cost Branching
SVM	Support Vector Machine
SVR	Support Vector Regression
T	Tie
TSP	Traveling Salesperson Problem
UCB1	Upper Confidence Bounds
UCT	Upper Confidence bounds for Trees
VS	Variability Scores

LIST OF APPENDICES

CHAPTER 1   INTRODUCTION

Mixed-Integer Programming (MIP) problems are optimization problems in which some decision variables represent discrete or indivisible choices. Formulations of this kind naturally model a number of real-world situations, making MIP a very successful paradigm to tackle decision-making tasks that arise from a wide variety of application domains – from transportation and energy, to healthcare and logistic operations.

Despite being $\mathcal{NP}$-hard problems, MIPs can nowadays be solved in reliable and effective ways: MIP computation witnessed dramatic advances over the last decades [1,2], and modern MIP solvers now tackle a wide range of instances by leveraging an extensive collection of both exact and heuristic algorithms. Notably, even exact methods employed to solve MIPs – among all, Branch and Bound (B&B) [3] – comprise a number of heuristic decisions in their design, to the extent that one could consider MIP solvers as systems that solve problems exactly while being intrinsically heuristic in nature [4]. In addition, many ideas from different domains (e.g., constraint programming, metaheuristics, satisfiability) have been successfully incorporated in MIP algorithms over the years [5], outlining an ongoing hybridization process of MIP techniques. As a result, the MIP algorithmic framework stands out today as a data-rich, interconnected and complex environment.

At the same time, the machine learning (ML) concept of "learning from examples" has been effectively leveraged in several domains to develop algorithms that learn from experience (i.e., from data, observations) how to perform a given task. While mathematical optimization intrinsically lies at the core of ML methods, recent years have also seen a rise in the application of learned predictions to optimization settings [6]. In particular, a fascinating question concerns the use of ML techniques in the algorithmic design of optimization solvers, independently of the application domain in which such algorithms are used.

The present thesis aims to explore this question, and focuses on applying ML to the MIP algorithmic framework. We see the ML paradigm as a way of complementing the capabilities of a general-purpose MIP solver and providing operational indications about structural decisions for which we lack in-depth understanding. In this sense, we believe ML algorithms could provide new tools for MIP computation, and lead it towards even more flexible and sophisticated outcomes.

The general goal of this research is thus to identify and analyze some original opportunities ML techniques may offer within the MIP algorithmic ecosystem, and subsequently develop and implement novel methods combining both technologies.

The rest of this chapter introduces basic definitions and concepts on MIP resolution methods and ML approaches, which are essential to discuss our research contributions and objectives.

1.1 Background and motivation

We define a general Mixed-Integer Program (MIP) as the optimization problem

$$z^* = \min\{f(x) : Ax \leq b, x_j \in \mathbb{Z} \ \forall j \in I\}, \tag{1.1}$$

where a real-valued objective function $f : \mathbb{R}^n \longrightarrow \mathbb{R}$ is minimized over variables $x \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$ and $I \subseteq N = \{1, \dots, n\}$ is the set of indices of variables that are required to be integral. We refer to $X_{MIP} = \{x \in \mathbb{R}^n : Ax \leq b, x_j \in \mathbb{Z} \ \forall j \in I\}$ as to the set of *feasible solutions*[1] of MIP (1.1). A feasible solution $x^* \in X_{MIP}$ is called *optimal* if $f(x^*) = z^*$.

When integrality requirements $x_j \in \mathbb{Z} \ \forall j \in I$ are dropped, one obtains the *continuous relaxation* of MIP (1.1),

$$z_P = \min\{f(x) : x \in X_P\}, \ X_P = \{x \in \mathbb{R}^n : Ax \leq b\}. \tag{1.2}$$

Note that

$$X_{MIP} \subseteq X_P \implies z_P \leq z^*, \tag{1.3}$$

i.e., the objective function value of the continuous relaxation (1.2) provides a lower bound to the optimal value of a MIP solution.

Different types of MIPs arise depending on the properties of the objective function (and, in general, also on the types of constraints, which we fix here as a system of linear inequalities $Ax \leq b$). In particular, when f is linear, $f(x) = c^T x$ for some $c \in \mathbb{R}^n$, one obtains a so-called Mixed-Integer *Linear* Program (MILP),

$$z^* = \min\{c^T x : Ax \leq b, x_j \in \mathbb{Z} \ \forall j \in I\}, \tag{1.4}$$

and its corresponding continuous relaxation is usually referred to as a *Linear Programming* (LP) relaxation. While a MILP is in general $\mathcal{NP}$-hard, its LP relaxation is polynomially solvable, and it is routinely dealt with efficiently in MIP solvers. Though most of the thesis focuses on MILPs, we will also encounter Mixed-Integer *Quadratic* Programs (MIQP), i.e., MIPs in which a quadratic objective function $f(x) = \frac{1}{2}x^T Q x + c^T x$, $Q \in \mathbb{R}^{n \times n}$, is minimized

---

[1]Being (1.1) a minimization problem, one should call *solution* to it a proper minimizer that also satisfies the constraints; however, the MIP community historically utilizes the term to identify all feasible points, i.e., the solutions of the satisfiability problem expressed by the constraints of (1.1).

under a set of linear constraints (see Chapter 4). The algorithmic framework for solving MIQPs significantly relies on MILP technology, which is (older and) overall more mature. We thus present in what follows the fundamental components of modern general-purpose MI(L)P solvers; for further details, the reader is referred to [7, 8].

1.1.1 Solving MILPs

The backbone algorithm of any modern MILP solver is the exact tree search method of Branch and Bound (B&B) [3]. Simply put, B&B follows a divide-and-conquer approach to partition the solution space of a MILP: *branching* refers to the operation of iteratively splitting the feasible region into smaller and easier-to-solve subproblems, each of which is mapped into a node of a decision tree. In addition, a *bounding* mechanism is used to prune unpromising branches from the search, avoiding the need of a full enumeration of the exponentially many solutions of a MILP and thus making B&B an *implicit* enumeration algorithm.

Starting with the initial MILP formulation (1.4) as the root node, at every node (including the root) integrality requirements are dropped and the LP relaxation is solved to obtain an optimal LP solution x^*. If x^* satisfies $x_j^* \in \mathbb{Z} \ \forall j \in I$ then it is also a feasible solution for the original MILP, and it provides an upper bound to the optimal value z^*. Otherwise, some variable with index in I must be violating the integrality requirements, and its *fractional* value in x^* can be used to split the variable's domain. More formally, $\mathcal{C} = \{i \in I : x_i^* \notin \mathbb{Z}\}$ defines the index set of fractional variables which are *candidates* for branching at a given node, after the LP relaxation is solved. The branching problem (*variable selection*) consists in selecting a variable $j \in \mathcal{C}$ to create child nodes according to the split

$$x_j \leq \lfloor x_j^* \rfloor \ \vee \ x_j \geq \lceil x_j^* \rceil. \tag{1.5}$$

Child nodes inherit a lower bound estimate from their parent, and (1.5) ensures that x^* is removed from their solution spaces (Figure 1.2a). Note that while the theoretical complexity of the created subproblems remains the same of the parent's one, the partition mechanism makes sub-MILPs smaller (and easier to solve) as the algorithm proceeds. After extending the tree by branching, the algorithm moves on to choose a new node from a list of *open nodes*, i.e., leaves in the search tree yet to be explored (*node selection*). A new relaxation is solved, and the exploration continues.

Maintaining global upper and lower bounds (called *incumbent* and *best bound*, respectively) and solving LP relaxations at each node allow the pruning of large portions of the search space. Specifically, node fathoming happens in three possible ways: by *integrality*, when the
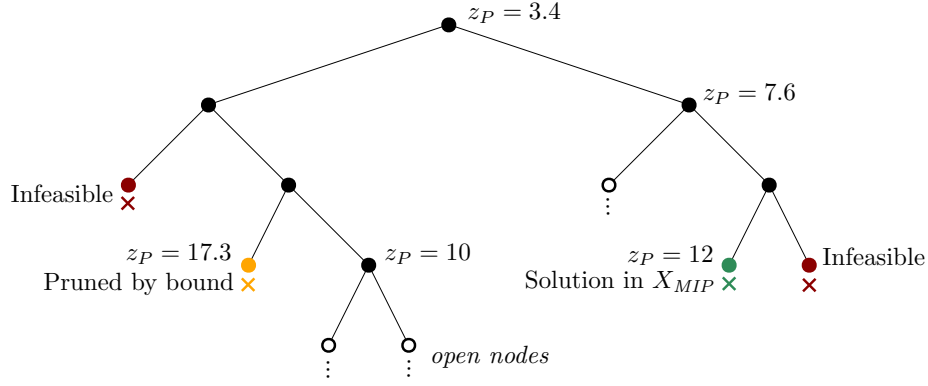
Figure 1.1 Example of a MILP branch-and-bound tree.

relaxed solution x^* is also feasible for (1.4); by *infeasibility* of the sub-problem, i.e., when no x^* is found; by *bound*, when the comparison of the node's lower bound to the incumbent proves that its sub-tree is not worth further exploration (Figure 1.1). At any point of the resolution process the *gap* between the global bounds allows to measure the quality of the present solution, and an optimality certificate is ultimately reached when the global bounds converge.

Besides applying the branching dichotomy (1.5) to rule out the LP solution x^* of a subproblem from future exploration, one can also try to tighten the relaxation itself by introducing *cutting planes* [9]. A cut is a linear inequality $\alpha^T x \leq \beta$ that separates x^* from the convex hull of X_{MIP} (see Figure 1.2b), in the sense that

$$\alpha^T x^* > \beta \ \text{ and } \ \alpha^T x \leq \beta \text{ is valid for (1.4).} \tag{1.6}$$

Even though a pure cutting plane algorithm is guaranteed to converge in a finite number of iterations [9–11], neither cuts nor branching are in practice used alone: on the one side, adding too many cuts can result in larger LPs and numerical issues for the solver; on the other side, LP relaxations are rarely good approximations of the convex hull of X_{MIP}, and some amount of strengthening (especially at the root node) can greatly help before and during the branching procedure. The resulting branch-and-cut (B&C) paradigm [12] is what general-purpose MILP solvers are nowadays based on.

Clearly, an incumbent found early on in the resolution process can significantly help to prune the B&C tree, and consequently reduce the number of nodes needed to prove optimality. Besides, sometimes a proof of optimality may not be computationally viable, and a good feasible solution becomes practically more important. Another valuable component of MILP solvers consists of *primal heuristics* [13], i.e., algorithms that aim to find (good) integer feasible so-
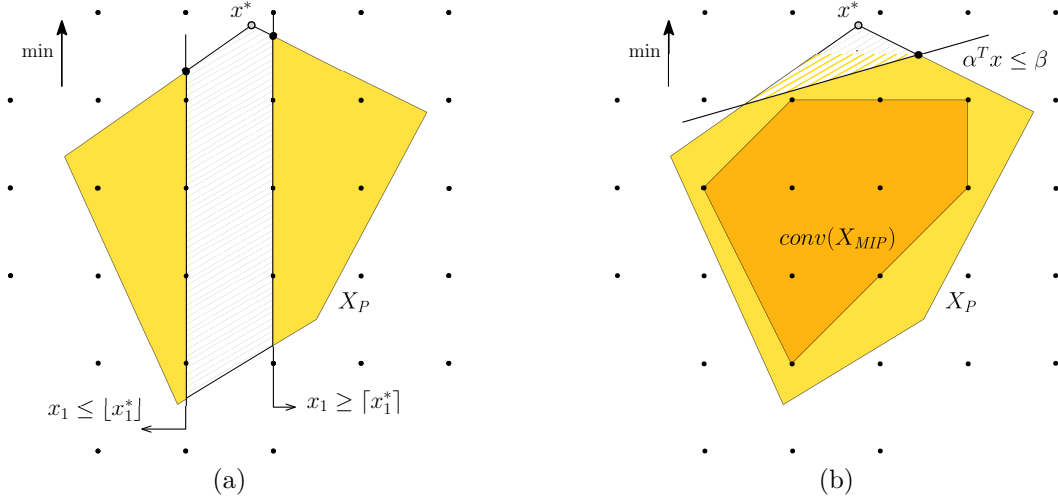
Figure 1.2 (a) Branching on the fractional value of variable x_1, from the solution x^* of the LP relaxation. (b) Separation of x^* from the convex hull of integer feasible solutions $conv(X_{MIP})$ via cutting plane.

lutions quickly – arguably the first goal one has in mind when starting to solve a MILP [14]. These procedures range from being very simple to very complex, and often incorporate concepts such as rounding, diving, and local search. Within MILP solvers, heuristics are usually run as periodic subroutines throughout the B&B search or as standalone procedures; paired with increasingly more sophisticated presolve and domain propagation steps [1, 5], they all contribute to the rich technology available today to tackle MILPs.

1.1.2 Machine learning in the picture

Even though general-purpose MIP solvers reached a mature and stable performance over the last decades, their algorithmic environment is anything but rigid: the flexibility of the MIP framework is highlighted in [4], and it is decisively linked to the fact that heuristic techniques and choices are everywhere in state-of-the-art algorithms. For example, the two crucial decisions of variable and node selection in B&B follow heuristic rules, which have been expertly tuned over many years and many computational studies (e.g., [5, 15]) to be effective across widely heterogeneous problem instances. When branching, though, a single bad decision at the beginning of the search could lead to no improvement and a doubled tree size, so employing clever branching schemes is critical for B&B success.

In some way, the intrinsically heuristic nature and the resulting flexibility of MIP algorithms can be viewed at the same time as a blessing and a curse for modern solvers: tie breaking mechanisms are far from perfect, and seemingly neutral changes (e.g., the order in which

variables are specified in the mathematical model) can have unpredictable consequences due to *performance variability* [16,17]. On the one hand, works like [18,19] showed that variability should not be viewed as an altogether bad phenomenon in MIP solving, and that it can be leveraged to stabilize and ultimately improve the solver performance. On the other hand, the sequential character of B&C and the tight connection between different solver components can produce unchecked cascade effects from individual heuristic decisions. An example of this is the choice of an LP basis among equivalent optimal ones at every relaxation, but in particular at the root node: which cuts are generated, which variable is branched on and which heuristics are applied, are all choices that depend to some extent from the selected initial LP basis. In some other situations, one may also lack theoretical understanding of more structural algorithmic decisions, as it might not be evident which solution method should be used or which formulation should be preferred to tackle a problem effectively.

Machine learning naturally lends itself as a tool to approach this type of concerns. As already mentioned, ML leverages the paradigm of *learning from examples* to infer a prediction rule, following a decision function that is acquired via optimization of a specific criterion. A little more formally, if we denote by $\mathcal{D} = \{\mathbf{z}_1, \ldots, \mathbf{z}_n\}$ a set of examples and by $\mathcal{F}$ a parameterized family of functions (e.g., that of all linear hyper-planes), the aim is to fit a map $f \in \mathcal{F}$ over points in $\mathcal{D}$, according to a loss functional $\mathcal{L} : \mathcal{F} \times \mathcal{D} \longrightarrow \mathbb{R}$ that evaluates the quality of the fitting. The ultimate ML goal is to find (*learn*) a predictive function that will *generalize*, i.e., perform well on future, unseen examples.

Different types of prediction targets characterize different ML tasks [20]. In *supervised* learning, examples are features-label pairs: features describe a sample in order for it to be qualitatively labeled among a number of categories (*classification*), or quantitatively assigned to a real value (*regression*). When targets are not specified in the data (*unsupervised* learning), the goal is to discover patterns, similarities and alternative representations for the samples. In *reinforcement* learning [21], instead, an agent has to learn from its own collected experience how to interact with an uncertain environment in order to maximize a reward function: perceiving information about the environment in the form of a *state* representation, the agent follows an internal policy to select the next state-changing *action*.

Learning process and predictive functions can utilize various techniques, ranging from traditional regression or tree-based models, to ensemble and Bayesian methods. More recently, there has been substantial effort in developing *deep learning* algorithms, which rely on artificial neural network architectures to process raw input features and automatically extract additional information that is valuable for the predictive task [22].

While a detailed introduction to ML and its paradigms can be found in Chapter 3 (Section

3.2), we note here how the ML approach could be leveraged in MIP computation, in particular in the algorithmic design of general-purpose solvers. Indeed, learned predictive models could be used where heuristic decision making is currently performed, to either refine or replace existing hard-coded rules, formulas and fixed algorithms' parameters. As we already observed, the MIP solving environment is inherently full of possibilities in this sense: variable and node selection, cutting planes and primal heuristics management are only some of the most intuitive options. Moreover, the integration of ML tools in the algorithmic system could involve different learning paradigms and approaches, and occur on various levels. For example, when tackling variable selection in B&B one could aim at mimicking existing branching schemes in a supervised way (e.g., with the goal of speeding up computations), or instead seek to learn how to flexibly utilize existing rules and adjust their parameters, or even learn an altogether new policy by reinforcement learning.

Libraries for MIP benchmarks have been growing their instance count (see, e.g., [23, 24]), and although gathering a dataset requires some careful considerations, so many decisions happen during the MIP optimization process that data itself – the foundation of ML approaches – is not missing. In fact, information from the resolution process is abundant yet mostly unexploited by state-of-the-art MIP solvers; in particular, it is not typically used in any statistical way to support decision making. While heuristic components are natural candidates for ML applications, outputs from the overall B&C optimization could be leveraged to identify trends and performance issues, monitoring (and, ideally, strategically conditioning) the search itself.

Finally, the combination of ML into MIP algorithmic design well fits into the ongoing hybridization of MIP (exact) techniques, which has proven to be a successful approach to develop more flexible and highly integrated solving environments.

1.2 Objectives and contributions

The general objective of this thesis is to explore opportunities in the application of statistical learning methods to MIP algorithmic design. Specifically, this dissertation contributes four works on the theme. The presented articles investigate different research questions and develop novel experimental frameworks to tackle them, thus adding to the growing literature of this recent research area, and extending its scope. The thesis is composed of self-contained chapters, which articulate the general research goal into four different ideas; for each of them we highlight their specific objectives as well as their achieved contributions.

On learning and branching: a survey (Chapter 3, [25])
Our first contribution consists of a survey documenting the early attempts in the application

of ML to the B&B framework, in particular those addressing the two decisions of variable and node selection. For each topic, we take the perspective of interpreting some fundamental MILP contributions in the light of ML approaches; we identify their concerns and assumptions, and show how recent ML-based works address (or dismiss) them. The motivation for this work comes from the need of organizing the initial ML contributions on the branching theme: by including them within a single framework, we aim to promote their understanding and discussion, as well as the identification of future research directions. To achieve that, the survey includes an introduction to ML and its paradigms, together with a discussion of transversal ML notions (e.g., feature engineering, generalization) that will be examined in the subsequent chapters as well.

A classifier to decide on the linearization of MIQPs in CPLEX (Chapter 4, [26, 27]) The second work tackles the question of whether to linearize the quadratic part of convex MIQPs when solving them with the IBM-CPLEX solver [28]: the decision deals with the reformulation of a MIQP during the preprocessing phase, and substantially conditions the downstream resolution algorithms employed by the solver. While CPLEX provides its users a switch parameter to decide on the linearization step, the decision is not clear cut: the goal of the paper is to use ML statistical tools to cast a prediction on this algorithmic choice. To pursue this objective, we frame the question *linearize vs. not linearize* as a classification one, and explore it by building a dataset of synthetic and real-world instances, proposing labeling schemes and carefully engineer features. The ML experiments and the evaluation metrics are designed to tightly integrate the optimization knowledge in the learning pipeline, and, as a result, a learned classifier is deployed in CPLEX (v12.10.0) to control MIQP linearization. These contributions allow us to establish the first example of an end-to-end integration of ML tools into a leading commercial solver: in this sense, the work also provides a methodological process for the combination of ML and MIP technology which we hope will serve as a reference in the area.

Learning MILP resolution outcomes before reaching time-limit (Chapter 5, [29]) The objective of our third contribution is to explore how information from the B&C optimization could be leveraged to identify trends and performance issues when solving MILPs. The work focuses on predicting, after only a fraction of the available computing time has passed, whether a generic MILP instance will be solved to optimality before time-limiting. More specifically, we use CPLEX to collect data from the first portion of the B&C tree, and apply feature-based sequence classification to predict a binary optimization outcome. Feature design is a key step of the process: the developed features attempt to describe the behavior of a MILP run in a quantitative way, taking into account the interplay of different

solver's components. While the current framework is limited to an offline setting, the work opens promising scenarios for MILP algorithmic design, as discovering early in the resolution process that a run will not terminate with a proof of optimality appears valuable for both MILP developers and end-users. The good predictions achieved also support the idea of there being a pattern in general-purpose B&C solving that is shared by heterogeneous MILP problems, and the fact that data from the optimization process could be used in a more structured, statistical way to get insights, trends, and ideally modify the solver behavior.

Parameterizing B&B search trees to learn branching policies (Chapter 6, [30])
The last presented work puts into practice what studied in [25] (Chapter 3) and contributes to the *learning to branch* theme. While most papers on the topic propose to imitate the strong branching rule and specialize it to distinct classes of problems, we aim instead at learning a branching policy that generalizes across heterogeneous MILPs. Building on the intuition gained from [29] (Chapter 5), our hypothesis is that parameterizing the state of the B&C tree can significantly aid this type of generalization. To explore the idea, we employ the SCIP [31] default branching scheme as the "expert" rule to be mimicked and propose a novel imitation learning framework. We contribute new input features as well as deep neural network architectures to represent variable selection; in particular, features contain a description of the status of the resolution process, and architectures can handle ever-changing inputs from the search tree. Experimental results clearly show that incorporating information from the B&C optimization leads on average to smaller explored trees and to better generalization properties of the learned branching policies. The developed paradigm appears to be promising: the idea and the benefits of parameterizing the search tree could be expanded even more in future experiments, especially in reinforcement learning settings.

1.3   Thesis outline

The remainder of this document is organized as follows. Chapter 2 contains a brief literature review on the application of ML to MIP, and provides useful pointers to related works. Chapters 3, 4, 5 and 6 form the main body of this thesis, and contain the four contributions outlined above. In particular, the survey on ML-based approaches for branching (Chapter 3) also includes a general introduction on ML paradigms and concepts, together with a review of state-of-the-art methods for variable and node selection in B&B; both contents are relevant for the rest of the chapters as well. In Chapter 7, we further discuss our contributions from a broader, unified perspective, and recognize the transversal themes that emerge in the different articles. Finally, Chapter 8 summarizes the dissertation work, comments on its limitations, and identifies future research directions and outlooks.

CHAPTER 2   RELATED WORKS

The idea of combining ML and discrete optimization is not new; indeed, the fields of Artificial Intelligence (AI) and Combinatorial Optimization (CO) have been influencing each other for decades, most notably through research in satifiability (SAT) and search algorithms. With ML emerging as a successful AI paradigm, the last few years witnessed a renewed interest in their integration: the probabilistic approaches of ML appeared promising in a variety of optimization settings, so the scope of the original AI-CO interaction has extended to new application domains and methodologies.

A considerable amount of work has been produced on the topic of algorithm configuration (see, e.g., [32,33]): complex optimization solvers usually employ a large number of parameters to control the interactions and executions of their algorithms, and parameters values can dramatically impact a solver's performance. Predictive methods have thus been applied to find good parametric configurations, and automatize the process of tuning solvers to special families of instances [34]. These problems are often studied together with that of runtime prediction; in [35], for example, runtime modeling techniques and features are discussed for SAT, MIP and travelling salesperson (TSP) instances.

The use of ML to complement and improve optimization solvers gained particular attention in MIP-specific settings. Given the prominence of B&B in the MIP algorithmic framework, it should not come as a surprise that the first applications of statistical learning tools in MIP concerned precisely B&B-related heuristic decisions. In particular, works on variable selection have been appearing at a steady pace over the last years (see, e.g., [36–40]), making *learning to branch* an established theme in the present literature. Other algorithmic tasks have also been addressed: node selection [41, 42]; the choice of running primal heuristics at B&B nodes [43]; the decision of whether a reformulation should be applied to a MIP, and which decomposition to choose among multiple available ones [44]. More recently, there has been some interest in learning to select cutting planes [45, 46].

Related to B&B explorations and to the already mentioned problem of runtime prediction is the idea of estimating the difficulty of an instance, for example by taking into account the size and shape of its explored search tree. While this type of prediction does not directly modify the optimization process (as instead does, e.g., variable selection), its practical impact is indeed relevant, especially when dealing with hard instances and limited computing resources. This topic has in fact interested both AI and optimization communities since the work of [47]; in the MIP context, we highlight the works of [48, 49] and [50].

The recent survey in [6] examines these and other works in a structured way, organizing them along methodological and algorithmic axes, and more generally discussing relevant issues that emerge when integrating ML into CO settings. Overall, the contributions that we mentioned so far employ various predictive models, and more generally combine ML and optimization according to different methodological paradigms; in addition, they also differ in terms of which types of problems they apply to (e.g., special combinatorial classes of instances or general MILPs). Despite their numerous differences, these works share the intention of using ML alongside optimization to improve or automatize some algorithmic decisions. Their predicted outcomes are embedded into exact algorithmic frameworks, with the integration happening on different levels that range from the isolated task of parameterizing an algorithm to repeatedly making decisions in solver's subroutines.

It is in this fruitful thread of research that the works of the present dissertation can be positioned. Broadly speaking, the classification question of Chapter 4 can be interpreted as a parameterization of the solver on the decision of linearizing convex MIQPs. On the topic of *learning to branch*, Chapters 3 and 6 contribute a survey on the theme and a novel experimental framework, respectively. Chapter 5, instead, deals with a more general prediction on the outcome of MILP optimization, which could be leveraged to guide the resolution process and better utilize computing resources.

The interplay between ML and discrete optimization can also be interpreted in alternative ways. Another explored approach is that of training ML models to heuristically solve MIP instances, e.g., by directly predicting their solutions. This paradigm has been used for designing heuristics algorithms for CO problems like TSP [51,52] and other optimization problems over graphs [53], where structure can be exploited to build solutions. Other works aim instead to predict solution-related outcomes: in the stochastic setting of [54] a neural network learns "tactical" aggregated solutions, while in [55] ML is used to estimate the value of optimized solutions, without practically running the optimization.

Conversely, ML and discrete optimization can be combined the other way round, i.e., with discrete optimization techniques being embedded in ML models. In [56,57], for example, MIP formulations are employed to learn optimal and explainable decision trees; in [58], instead, the column generation method is leveraged to search over exponentially many clauses when learning a Boolean rule. Finally, the works of [59, 60] embed optimization in deep learning frameworks by conceiving CO problems as distinctive neural network layers.

CHAPTER 3   ARTICLE 1 – ON LEARNING AND BRANCHING: A SURVEY

Authors: Andrea Lodi and Giulia Zarpellon[1]

Published in *TOP*, 2017.[2]

Abstract This paper surveys learning techniques to deal with the two most crucial decisions in the branch-and-bound algorithm for Mixed-Integer Linear Programming, namely variable and node selections. Because of the lack of deep mathematical understanding on those decisions, the classical and vast literature in the field is inherently based on computational studies and heuristic, often problem-specific, strategies. We will both interpret some of those early contributions in the light of modern (machine) learning techniques, and give the details of the recent algorithms that instead explicitly incorporate machine learning paradigms.

3.1   Introduction

In the last decade we have experienced the impressive development of powerful artificial intelligence algorithms able to perform complex tasks in the form of so-called "predictions" in contexts as diverse as image recognition, natural language interpretation, word alignment, etc. Those algorithms are not only theoretical but, in addition, they have been effectively deployed into reliable software packages commonly used by all sort of intelligent devices, computers, sensors, smart TVs, and smart phones. This revolution, whose potential is not yet fully understood and not yet fully realized, has been possible because of two main ingredients. On the one side, the impressive increase of computing power (especially Graphical Processing Units) paired with the enormously extended technological (hardware and software) capability of collecting data, often in the form of examples. On the other side, the shift by (part of) the artificial intelligence community, that of machine learning (ML, in short), of the learning paradigm from "knowledge formalization" to "learning by examples", which enables perception and a form of learned intuition.

Of course, that stream of success has attracted the attention not only of the business world but also of other scientific communities that became interested in exploring the possibility of using ML techniques within their algorithms and methods, to be able to tackle structural challenges that have resisted traditional approaches. Clearly, this applies to the context of

---

[1]Authors are listed alphabetically, as is standard practice in Operations Research journals and conferences.

[2]Available at [25].

using ML algorithms within domain applications as healthcare, transportation, energy, and virtually anywhere a knowledge acquisition is required by the decision-making process.

However, for mathematical optimization, one of the most fascinating question concerns the use of ML techniques in the algorithmic design, independently of the application domain in which optimization algorithms are used. This question can be asked in many contexts and it is certainly related to the fact that breaking ties in optimization algorithms is far from perfect, see, e.g., [16] and [17]. Indeed, learning mechanisms able to discover structural properties of seemingly equivalent components of an algorithm would certainly be very useful. For example, given the *Mixed-Integer Linear Programming problem* (MILP)

$$\min\{c^T x : Ax \geq b, x \geq 0, \ x_i \in \mathbb{Z} \ \forall i \in I\}, \tag{3.1}$$

learning a "better" initial basis among the equivalent (optimal) ones of the *linear programming* (LP) relaxation

$$\min\{c^T x : Ax \geq b, x \geq 0\}, \tag{3.2}$$

could lead to a reduction of the so-called *performance variability* (see again [17]) as well as be beneficial from the performance standpoint (see, e.g., [19]).

In this survey, we concentrate on the questions, within such a flavor that we can call *learning for optimization*, that can be asked concerning the crucial decisions of the most well-known exact method for discrete optimization, i.e., the *branch-and-bound* algorithm [3].

In its basic version, the branch-and-bound algorithm iteratively partitions the solution space into sub-MILPs (the children nodes) which have the same theoretical complexity of the originating MILP (the father node, or the root node if it is the initial MILP). For MILP solvers, branching usually creates two children by using the rounding of the solution of the LP relaxation value of a fractional variable, say x_ℓ, constrained to be integral, $\ell \in I$,

$$x_\ell \leq \lfloor x_\ell^* \rfloor \quad \vee \quad x_\ell \geq \lceil x_\ell^* \rceil, \tag{3.3}$$

where x^* denotes the optimal solution of (3.2). The two children above are often referred to as *left* (or "down") branch and *right* (or "up") branch, respectively. On each of the sub-MILPs the integrality requirement on the variables $x_i, \forall i \in I$ is relaxed and the LP relaxation is solved. Despite the theoretical complexity, the sub-MILPs become smaller and smaller due to the partition mechanism (basically, some of the decisions are taken) and eventually the LP relaxation is integral for all the variables in I. In addition, the LP relaxation is solved at every node to decide if the node itself is worthwhile to be further partitioned: if the LP

relaxation value is already not smaller than the best feasible solution encountered so far, called *incumbent*, the node can safely be fathomed because none of its children will yield a better solution than the incumbent. Finally, a node is also fathomed if its LP relaxation is infeasible.

Paired with the *cutting plane* algorithm [9] to obtain variations of the *branch-and-cut* paradigm [12], the branch and bound is the most basic structural component of modern MILP solvers (see, e.g., [2, 61]). As pointed out in [4], the *exact* computation performed by MILP solvers relies on a somehow surprising collection of *heuristic* decisions, two of the most crucial ones being those associated with the branching scheme outlined above, namely

- *variable selection*, which of the variables x_ℓ, $\ell \in I$ among those fractional at any node, to branch on in (3.3), and

- *node selection*, which of the currently open nodes to process next.

In Section 3.3 the most traditional and effective strategies to deal with the two decisions above are discussed. This survey documents the recent attempts to incorporate sophisticated learning mechanisms within those strategies. In order to do that, we present in Section 3.2 a brief overview of the machine learning concepts that are required to understand the algorithms surveyed in Sections 3.4 and 3.5. Finally, Section 3.6 discusses a few additional references related to learning within a branching tree and draws some conclusions outlining some possible research directions.

3.2   A brief overview of machine learning

As already mentioned in Section 3.1, machine learning is the subfield of artificial intelligence devoted to develop intelligent systems that learn from experience (i.e., from examples, or observations) how to perform a given task. In ML, the prediction process is performed in an operational way, using information coming from data and following some specified criterion; optimization is undoubtedly one of the cores of this process.

The aim of this section is to make the reader familiar with the essential concepts of learning: we will introduce the traditional learning framework and its standard tasks of supervised and unsupervised learning in Section 3.2.1. We will then mention in Section 3.2.2 some issues and pitfalls of learning that every researcher using ML should be aware of while developing an application, and we will conclude the section with a brief introduction to another learning paradigm called reinforcement learning in Section 3.2.3.

For more details and material we refer to [20–22, 62].

3.2.1 The standard learning setting and tasks

We can formalize the traditional learning framework starting with a set of n examples $\mathcal{D}_n = \{\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_n\}$, where we denote with $\mathbf{z}_i$ the realization of a random variable Z_i under an unknown process $P(Z)$. Each $\mathbf{z}_i$ is supposed to be independently drawn from the distribution specified by $P(Z)$.

One wishes to learn a function f exploiting some characteristics of $\mathcal{D}_n$ (and hence $P(Z)$), with the aim of employing it to make future predictions on new examples. The search for f is performed among the members of a certain (parametrized) family of functions $\mathcal{F}$. Together with $\mathcal{F}$, a loss functional $\mathcal{L} : (\mathcal{F}, \mathcal{D}_n) \to \mathbb{R}$ is specified in order to evaluate the quality of different available $f \in \mathcal{F}$.

The ultimate goal would be to find f^* as a minimizer of $\mathbb{E}[\mathcal{L}(f, Z)]$, the expected value of $\mathcal{L}(f, Z)$ under $P(Z)$, following the so-called *expected risk minimization* principle. However, being compelled to work in a restricted space of functions $\mathcal{F}$ and not knowing $P(Z)$ *a priori*, one aims instead at minimizing the *empirical risk*, using the examples of the finite set $\mathcal{D}_n$ to learn a function

$$f_{\mathcal{D}_n} \in \mathrm{argmin}_{f \in \mathcal{F}} \ \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(f, \mathbf{z}_i), \tag{3.4}$$

which will then be employed as a predictor.

The examples $\mathbf{z} \in \mathcal{D}_n$, the loss functional $\mathcal{L}$ and the prediction goal of the to-be-learned function $f \in \mathcal{F}$ can be various: such differences in forms and intents are what characterize different ML tasks. Not pretending to be exhaustive, we will briefly go through two main types of learning: supervised and unsupervised ones. The intent is to familiarize with the general framework, in order to properly locate the algorithms that will be considered later on.

Supervised learning In supervised learning, data in $\mathcal{D}_n$ consist of pairs $\mathbf{z} = (\mathbf{x}, \mathbf{y})$; we call $\mathbf{x} \in \mathbb{R}^d$ the *input* or *features* vector, and $\mathbf{y}$ the *output* or *target*. Depending on the output type, two main predictive tasks can be identified:

- Classification: the target is a qualitative label, used to differentiate between $m \in \mathbb{N}$ classes or categories. The scalar label $y \in \{1, \ldots, m\} \subset \mathbb{Z}$ can be encoded in $\mathbf{y} \in \mathbb{Z}^m$, and one could for example choose to measure the accuracy of a classifier f by means of the classification *error rate*: often, the *expected 0-1 loss* is evaluated by considering the loss of an example as $\mathcal{L}(f, (\mathbf{x}, \mathbf{y})) = I_{\{f(\mathbf{x}) \neq \mathbf{y}\}}$, where $I_{\{\cdot\}}$ denotes the indicator function.

- Regression: the target is a quantitative output $\mathbf{y} \in \mathbb{R}^m$; the prediction $f(\mathbf{x}) \in \mathbb{R}^m$ of

a regressor f estimates the expected value of $\mathbf{y}$ given $\mathbf{x}$. An example of loss functional commonly used in this setting is the *quadratic error* $\mathcal{L}(f, (\mathbf{x}, \mathbf{y})) = \|f(\mathbf{x}) - \mathbf{y}\|^2$.

Unsupervised learning   As the name suggests, in the paradigm of unsupervised learning the prediction is performed without a "supervisor" knowing the correct answers: data does not come with a specified target, but only as input of features $\mathbf{z} = \mathbf{x} \in \mathbb{R}^d$. The aim is to learn a function f describing in some way the unknown process $P(Z)$ from which the examples were drawn. Some common tasks in this setting are:

- Density estimation: f is an estimator of the distribution of input data. Since the concept of error rate is not suited for this unlabeled context, one could aim at maximizing the *(log-)likelihood* of the observed data, i.e., their probability with respect to the underlying distribution $P(Z)$.

- Clustering: the purpose of the learning is to discover similarities within the input space. Data can be grouped in hard-cut clusters or within a soft partition of the space; for a new point one could predict its memberships to one or several groups.

- Dimensionality reduction: a new representation of the input data is constructed, usually in a lower dimensional sub-space of the original input space (aiming at data visualization, for example). The goal is to identify some important characteristics of the input $\mathbf{x}$, e.g., via selection or extraction.

Among the numerous resources available for implementing ML algorithms, it is worth mentioning few open-source libraries, easily accessible by not-necessarily-expert users. For example, scikit-learn [63], mlr [64], and MachineLearning [65], which are built on the commonly used ML programming languages Python, R, and Julia, respectively.

3.2.2   Few things to keep in mind

Undertaking a project involving ML can be a non-smooth path to pursue, especially for beginner practitioners with a non ML-related background. The purpose of this subsection is to warn the reader against some non-trivial issues that could easily be encountered along the way.

The extensive discussion of those issues and the methods to overcome them is beyond the scope of this paper. Instead, we point out the importance of performing learning with awareness, i.e., by following the best practices of the field. An interesting outlook of these and other concerns can be found in [66].

Features engineering   A key step in every ML application is the design of what the input data represent, i.e., what are the shape and the meaning of the examples $\mathbf{z}$ that the learned function $f_{\mathcal{D}_n}$ should be receiving as argument in order to make a prediction. Often, one would pre-process raw data $\mathbf{x}$ into $\phi(\mathbf{x})$ by means of a features extraction procedure, in order to find a better suited representation of the problem, to be fed to a learning algorithm with. With a minor abuse of notation, we will treat input and features indifferently, denoting them as $\mathbf{x}$ or $\phi(\mathbf{x})$ or ϕ, depending on the context.

The features vector $\mathbf{x}$ encoded in the sample $\mathbf{z}$ should represent the problem at hand accurately: features are domain-specific and require some *a priori* knowledge about the nature of the problem, thus they are most often manually designed. Systems like those used in *Deep Learning* [22] are able to learn valuable features automatically, but the human intervention is still needed in the task of tuning the resulting complex architecture. In learning, it is possible for a single feature to be irrelevant if considered alone, and to become very significant when combined with other traits.

While the intuition suggests that the more features one has, the more information one will gain and the better the prediction will be, things are not as trivial. The risk is to incur in the so-called *curse of dimensionality* [67]: having many features corresponds to working in a high-dimensional space, where a limited dataset could result in a very sparse sampling, and the locally-based assumption that similar examples lead to similar predictions might fail.

Anyhow, features engineering is an aspect of learning that requires care and an iterative trial process, during which some features could be added, some discarded, some others combined, without a precise recipe to follow.

Generalization, overfitting and model selection   The ability to achieve *generalization* (i.e., the ability to perform well on previously unseen examples) is the fundamental property to look for in a learned predictor. However, the fact that one is bound to search within $\mathcal{F}$, and the availability in $\mathcal{D}_n$ of only a finite number of observations, make the learning of an optimal predictor a delicate task. One of the most frequent trap when dealing with learning is *overfitting.*

Broadly speaking, the phenomenon of overfitting represents the inability of a learned function to generalize: this occurs because $f_{\mathcal{D}_n}$ as in (3.4) is optimized to fit the specific dataset $\mathcal{D}_n$, making $\frac{1}{n}\sum_{i=1}^{n}\mathcal{L}(f_{\mathcal{D}_n},\mathbf{z}_i)$ an underestimate of the expected risk (also called *generalization error*), which is computed on new never-seen examples. In particular, the more the learned function fits the data in $\mathcal{D}_n$, the more the generalization error will be optimistic. Overfitting is strictly related to the choice of *model complexity* and the so-called *bias-variance trade-off*,

and makes it necessary to use some caution when establishing the "best" learning model among many.

In order to compare the performances of different learning models, a proper model selection procedure prescribes to divide the dataset into three parts, to be used in distinct *training*, *validation* and *test* phases. Different hyper-parameters versions of a chosen model are considered; each version's parameters are optimized during the training phase on the first share of the dataset. The learned models are then ranked with respect to their predicting performance on the validation set, and the hyperparameters setting resulting in the best performance is selected. Finally, to determine the generalization performance of the ultimately chosen model, a neutral test set is employed; examples that were never faced before are used in order to avoid a biased, optimistic estimation.

For further details on overfitting, standard procedures to avoid it and additional references, we refer to [62], ch. 7.

3.2.3 Another paradigm: reinforcement learning

We introduce now a third type of learning scheme, deviating from the traditional supervised/unsupervised settings we outlined in Section 3.2.1. *Reinforcement learning* is concerned with how an agent learns while interacting with an uncertain environment in order to maximize its reward. The agent is able to perceive some information about the state of the system it lives in, and can take state-changing actions that result into a *reward signal* (or a punishment one), evaluating its behavior. Each action affects later inputs of the system, and hence subsequent rewards; the goal is that of finding a policy maximizing the long-term return.

The setting of a Markov decision process to be optimally controlled provides a (partial) theoretical framework for reinforcement learning. At discrete time steps $t = 0, 1, 2, \ldots$, the agent observes the environment state $s_t \in \mathcal{S}$, and subsequently takes an action $a_t \in \mathcal{A}(s_t)$ with probability $\pi(a_t|s_t)$. The state changes into s_{t+1} with probability $P_{s_t,s_{t+1}}^{a_t}$, and an immediate reward $R_{s_t,s_{t+1}}^{a_t}$ is received. The goal is to learn an optimal policy $\pi : \mathcal{S} \to \mathcal{A}$ typically maximizing the expected long-term *discounted return*

$$G(s_0, \pi) = \mathbb{E}[R_{s_0,s_1}^{a_0} + \gamma \, R_{s_1,s_2}^{a_1} + \cdots + \gamma^t \, R_{s_t,s_{t+1}}^{a_t} + \ldots], \tag{3.5}$$

where $\gamma \in [0, 1)$ is the discount rate. The concept of discounting is useful in this framework to model the fact that a future reward is worth less than an immediate one. In particular, if $\gamma = 0$, the impact of future actions is not taken into account, i.e., the agent is short sighted

and maximizes at each step the immediate reward only, potentially reducing its total gain. Usually, $G(s_0, \pi)$ is referred to as the *value* of state s_0 following the policy π.

A classical reference for reinforcement learning is [21]; additional material can be found in [68] and [69].

Note that this third paradigm is inherently different from the previous two: while in supervised learning one is provided with examples correctly labeled with their "optimal" response, in reinforcement learning an agent has to learn from its own experience what a good behavior is, by a process of trial and error. Moreover, though an agent indeed discovers some implicit characteristics of its environment, the ultimate goal of reinforcement learning is that of maximizing a reward, and not that of finding hidden structures within the unlabeled examples, as it is instead for unsupervised learning.

A peculiar concern of reinforcement learning is the trade-off between *exploration* and *exploitation*. On the one hand, the agent should explore the space by trying new actions to know if they are valuable, whereas, on the other hand, it should exploit those actions yielding a high reward that it has already experienced. Any kind of unbalanced learning will produce poor results.

3.3 The branch-and-bound framework

As anticipated in Section 3.1, the variable and node selections are largely seen as the most crucial decisions in exact methods for MILP. On the one hand, branching on a variable that does not lead to any serious simplifications on any of the (two) children can be seen as doubling the size of the tree with no improvement, thus leading to extremely large (out of control) search trees. On the other hand, effective MILP solvers need to provide a good compromise between finding good solutions quickly and the chance of proving optimality in the short-to-medium term, a trade-off which is of course related to the way the search tree is explored.

Variable selection problem This is the task of deciding how to partition the current node, i.e., on which variable to branch on in order to create the two children. For a long time, a classical choice has been branching on the most fractional variable, i.e., in the 0-1 case the closest to 0.5 (sometimes referred to as *most infeasible branching*, MIB in short). That rule has been computationally shown in [70] to be worse than a complete random choice. However, it is of course very easy to evaluate. In order to devise stronger criteria one has to do much more work. The extreme is the so called *strong branching* (SB, in short) technique

(see, e.g., [15, 71]). In its full version (FSB, in short), at any node one has to tentatively branch on each candidate fractional variable and select the one on which the increase in the bound on the left branch times the one on the right branch is the maximum. Of course, this is generally very time consuming but its computational effort can be easily limited in two ways: on the one side, one can define a much smaller candidate set of variables to branch on and, on the other hand, can limit to a fixed (small) amount the number of simplex pivots to be performed in the variable evaluation. Another technique is *pseudocost branching* (PC, in short) which goes back to [72] and keeps a history of the success (in terms of the change in the LP relaxation value) of the branchings already performed on each variable as an indication of the quality of the variable itself. Among the most recent effective and sophisticated methods, *reliability branching* (RB, in short) [70] integrates strong and pseudocost branchings. The idea is to define a reliability threshold, i.e., a level below which the information of the pseudocosts is not considered accurate enough and some strong branching is performed. Such a threshold is mainly associated with the number of previous branching decisions that involved the variable. Finally, *hybrid branching* [73] computes for each candidate variable five different measures, chosen among typical branching scores of MILP, constraint satisfaction and satisfiability technologies. The measures are first normalized and then combined into a single score by means of a weighted sum.

Node selection problem   This is the most classical task of deciding how to explore the tree: one extreme is the so called *best-first* strategy in which one always considers the most promising node, i.e., the one with the smallest LP value, while the other extreme is *depth-first* where one goes deeper and deeper in the tree and starts backtracking only once a node is fathomed, i.e., it is either (mixed-)integer feasible, or LP infeasible or it has a lower bound not better (smaller) than the incumbent. The pros and cons of each strategy are well known: the former explores less nodes but generally maintains a larger tree in terms of memory, while the latter can explode in terms of nodes and it can, in case some bad decisions are taken at the beginning, explore useless portions of the tree itself. All other techniques, more or less sophisticated, are basically hybrids around these two ideas, like interleaving best-first and diving, i.e., a sequence of branchings without backtracking, in an appropriate way. Some of those techniques are discussed in Section 3.5.1.

Besides the fact that the above decisions are highly crucial for the effectiveness of the branch-and-bound (B&B, in short) framework and, in general, for the MILP technology, the urge of trying to use sophisticated learning mechanisms to guide them is motivated by their poor understanding from the mathematical standpoint. As far as we know, the only attempt in

the direction of formally studying the variable branching problem is the recent work of [74]. The authors, recognizing the limitations of traditional branching rules, propose an abstract and analytically tractable model for branching. Under few assumptions, they are able to establish new branching rules that appear to be effective for MILP resolution. In general, though, there is no deep understanding of the theory underneath branching, if any exists, so the application of (modern) statistical methods seems quite appealing.

The next two sections define the core of the present survey by systematically introducing ML approaches to variable selection, Section 3.4, and node selection, Section 3.5.

3.4   Variable branching heuristic

In general terms, all branching rules aim at guiding the search of the B&B tree in an efficient way, by appropriately choosing at each node the fractional variable one ought to branch on. At every step of the search, plenty of heterogeneous information can be gained, and a meaningful branching strategy should integrate and exploit this by updating knowledge in its decision-making rule.

The common denominator of the papers discussed in this section is their attempt to define a branching strategy extracting novel kinds of information and combining them in original ways. In the most recent works, the aggregation of the collected information is performed by means of ML algorithms. Nonetheless, we present as well some early works not mentioning the ML framework, which we could consider precursors in conveying the idea of gathering more and diverse data (in a ML framework, one would call them features) to capture the state of the B&B system and improve its decision-making process.

As observed in [75], the idea of extracting some characteristics to derive a branching rule is indeed what traditional heuristic schemes for branching already perform: fractionalities are employed in most-fractional branching; LP gains, measuring the impact that a candidate variable could have on the objective function value, are computed in SB and estimated with pseudocosts, taking into account historical data of the search.

In the recent works we will discuss, the novel trait is that of exploiting (possibly a large quantity of) collected data, and employing the learning framework to come up with more informed and complex decision functions, estimating a good branching strategy.

With the underlying belief that a more sophisticated and high-performing branching rule could be detected, the following works explore different pertaining questions: Which information should be used? How could it be efficiently extracted and appropriately learned? Which criteria should guide the search?

The next subsection is devoted to the discussion of some "forerunner" works (see Section 3.4.1). We will see how they already implied shared views on the role of learning for branching, on which recent ML-based attempts (treated in Section 3.4.2) are currently building up.

3.4.1 Precursors of "learning-to-branch"

The possibility of improving the general B&B performance by collecting and exploiting more information than customary is already questioned in [76]. The authors investigate whether typical B&B information extracted *two* levels deeper than a given node would influence the decision of the branching variable. The devised *lookahead branching* rule aims at maximizing both bound improvement and node pruning, and the gathered additional information proves to be useful not only for the purpose of defining a new branching rule, but also for auxiliary tasks such as bound fixing and simple implications deduction.

Clearly, the computational effort of performing such a forward scouting is significant. Hence, an abbreviated (and cheaper) version of the scheme is also defined. Even though the total number of explored nodes is reduced in certain instances if compared with a classical one-step lookahead SB implementation, the authors themselves point out that lookahead branching is too costly to be likely employed as a default branching scheme for MILP. However, it might be that for some classes of problems one might be willing to afford some supplementary computational cost, if the effort could somehow pay off.

As many other traditional heuristics, lookahead branching relies on LP gains to measure the impact of a candidate variable with respect to the collected additional information. A different point of view is developed in [77], where the measure of impact is based instead on fathoming decisions. This choice is motivated by two broadly acknowledged assumptions:

(*i*) the final goal in branching is minimizing the total number of explored nodes; in this sense, a *node-efficient* method is sought;

(*ii*) branching decisions are more crucial at the top of the tree.

The paper investigates the idea of a three-phase method for binary MILP problems. Specifically, a *clause* is defined as a partial assignment of the binary variables that cannot lead to possible improvements of the objective function. The authors exploit the fact that in any binary B&B tree a fathomed node gives a clause, and that this kind of information can be further strengthened to yield more fathoming. First, a given instance is partially solved: within

an upfront *collection phase*, clauses associated with fathomed nodes of the incomplete B&B tree are gathered, until their number reaches a threshold (fixed at 200). In the subsequent *improvement phase* the basic information is refined by solving an auxiliary MILP problem, in order to be finally employed in the *restart phase*, when the instance is fully solved with the gained information.

Some fathoming-based branching rules are defined, taking into account various possibilities for assigning weights to the collected clauses and estimating the effect of fixing and branching on a candidate variable, in a fashion that reminds combination rules for pseudocosts (see, e.g., [15]). Moreover, the improved information is exploited for two additional tasks. The combination of fathomed-based branching with *cuts generation* and a sort of *propagation* yield improved performance with respect to CPLEX [28] (version 11.1) with and without dynamic search.

However, there is no clear winner among the tested strategies, and it is not obvious to identify the benefits of using such improved information for branching. Indeed, more information associated with clauses could potentially result in longer computing times. Having in mind a ML framework, one could think of the proposed collection phase as a kind of training phase personalized (and repeated) for each instance, where the clause information themselves (namely, the features) are instance-specific.

Partially following the work of [77] is the *backdoor branching* approach in [78]. In this work, the collected data include some fractional solutions that are characterized as hurdles for the LP gap reduction. The learning phase is in fact a *sampling phase*, consisting of a multiple restart scheme. Iteratively, a given problem is partly solved until a certain number $K_{max}(= 10)$ of fractional solutions are encountered. The collected fractionalities become the input of a set covering model, aimed at computing a minimum cardinality *backdoor*, i.e., a minimum set S of branching variables whose integrality is enough to ensure that a certified optimal solution value is reached. Variables in S are assigned a priority to be chosen for branching, and a new updated incomplete run is performed. After $R(= 10)$ iterations, or when a backdoor S with $|S| > \Gamma(= 5)$ is found, a final *long run* is executed, employing a MILP solver as a black-box to which only the latest priorities are specified.

The method is compared with two settings of CPLEX (version 12.2): the default solver (with no cutoff provided), and a variant sharing the same setting as the tested algorithm, in which the optimal solution is provided, and cuts, heuristics and variable aggregation in preprocessing are deactivated. Backdoor branching compares well with the competitor method sharing its similar setting, and turns out to be very helpful in searching the top levels of the tree effectively.

Note that the purpose of the designed *backdoor branching* procedure is not that of selecting a single variable by means of a score. Instead, the goal consists of identifying a subset of variables that are in some sense top-ranked with respect to a certain branching priority measure; a similar idea will be encountered later on. The authors briefly mention the trade-off between collecting (more) reliable information and the cost coming with it. Two nice-to-have properties of prospective features can clearly be outlined.

- Features should be *relevant*, i.e., they ought to precisely and (as far as possible) completely describe those aspects of the B&B system playing a key role in the optimization and its efficiency.

- Features should be *low-cost* to compute, i.e., they should not constitute a computational burden.

The latter property could actually be drawn from a more general assumption, legitimate in the definition of an effective branching:

(*iii*) while being node-efficient, a good branching scheme is *time-efficient* as well.

Note that in backdoor branching, as in [77], the learning phase is actually a sampling phase that needs to be repeated for each instance, thus not qualifying as a learning-to-generalize mechanism. For each problem, shallow explorations of the search tree are performed, in order to figure out a likely good path in the final run. Finally, observe that in both [77] and [78] the collected information is manipulated by (optimally) solving MILP or LP problems, somehow learning by relying only on MILP technology.

A completely different paradigm is explored in [79], where a typical AI tool such as Information Theory [80] is used to derive branching rules. The approach is motivated by the interpretation of the B&B tree evolution as a search process, carrying certain and uncertain information. The essential observation is that nodes at the beginning of the search hold a high amount of uncertainty about the values of the variables, while at the end of the tree there is no uncertainty of this kind left. The idea is hence to guide the search in order to remove uncertainty, or, in other words, to propagate as much as possible the sure information carried by variables. In practice, the fractional values of integer-constrained variables, i.e., the basic feature of the most-fractional branching rule, are treated as probabilities, indicating the confidence in expecting the variable to be greater than or equal to its current value in the optimal solution. To measure the amount of uncertainty/information of a variable, the notion of *entropy* [80] is employed.

Four diverse families of *entropy branching* (EB, in short) heuristics are designed. The first family performs branching with one-step lookahead as in SB but choosing the variable yielding children with smallest uncertainty. Note that EB alone does not employ the objective function in taking a branching decision. The second family explores hybrid approaches: SB and EB scores are combined in terms of pure ranking positions or by means of a weighted sum. Additionally, EB is tested as tie-breaker for the classical SB rule. A third family includes methods that do not perform lookahead, but use instead the LP values of a current solution; two rules are defined in the special contexts of combinatorial procurement auctions and facility location problems (see again [79]). The last proposed family deals with multi-variables branches, considering branching on the sum (of values) of a subset of variables. The purpose is that of selecting the set of variables to be branched on as that resulting in the smallest entropy in a one-step lookahead.

Computational experiments performed on MIPLIB 3.0 [81] show no clear winner between SB and EB for the defined rules, while EB outperforms SB on some hard real-world procurement instances. Apart from the computational results, the authors' high-level discussion on branching approaches promotes further thinking about the various kinds of information that should be employed in a good branching strategy. In their point of view, different strategies refer to the different ways one could try to reach the goal of node-efficiency in B&B. Since a path in the tree can end in three possible ways, see Section 3.1, the authors try to interpret the existing branching methods with respect to the concurrent goals of driving the search towards early fathoming, early feasibility and early integrality.

We conclude the overview of forerunner works with [36], where a more framed use of ML techniques for variable branching heuristics is implemented. Slightly detaching itself from the others, this work does not aim at finding a new branching rule, but instead at best combining some existing ones along the tree. The motivating background is that of *portfolio algorithms*, where given a set of different methods one wants to predict and use the best available method with respect to the instance to be solved. Such techniques are based on the observation that it is unlikely that it does exist a single method dominating all others for every instance, and hence one looks for a dynamic procedure able to behave with flexibility, in an instance-specific way.

The idea in [36] is to devise an algorithm dynamically switching between different branching heuristics along the branching tree, choosing among them with respect to the different encountered subproblems. The goal of *DASH* (*Dynamic Approach for Switching Heuristics*) is precisely that of guiding the search by selecting the best branching rule, following the changing state of the B&B system. A similar dynamic scheme was introduced in [82] for the

special case of set partitioning problems.

In [36], the authors define a features space comprising 40 traits that capture aspects of the subproblem regarding its MILP formulation as well as its position in the B&B tree. As sought, features computation does not constitute an expensive overhead. A portfolio of six traditional branching heuristics is implemented: most(less) fractional rounding, most(less) fractional and highest objective rounding, pseudocosts branching with weighted score, and product score. Finally, a dataset of 341 instances coming from heterogeneous benchmark sets is considered, and split into a training and a test set. The first learning step is a clustering of problems motivated by the assumption according to which problems with similar features will yield to the same chosen heuristic. The grouping of instances is carried out by the *g-means* algorithm [83], a method similar to the classical *k-means* [84], which additionally determines in an automatic way the optimal number of clusters, assuming an underlying Gaussian density distribution. In particular, an extended training set is provided for clustering, in which some computed subproblems of the original training instances are added to the training set itself, with the intent of making the dataset more representative of the subproblems in the tree. Once the clusters are identified, a "best" heuristic is assigned to each of them. Clearly, this assignment is delicate and qualifies as a key component of the entire procedure: given the continuous changes of the subproblems types along the branching tree (partly due to the chosen branching heuristic itself!), the assignment needs to be performed simultaneously for all clusters. The step is undertaken by the parameter tuner GGA [85], and only the original (i.e., not extended) training set is employed.

Once the training setup is completed, at a given node of the tree, DASH computes the features of the subproblem and its nearer cluster (the Euclidean distance with respect to the clusters' centers is measured), and subsequently employs the assigned "best" branching heuristic for the selected cluster. In practice, switches do not happen at every node. This enforced limitation is motivated by the wish of further reducing the computational cost of features extraction, and by the fact that features change progressively along the tree, as it is shown by a 2-dimensional *Principal Component Analysis (PCA)* [86]. As a consequence, the switch is activated only up to the 10th depth-level of the tree and just at some prefixed points in time (every 3rd node); in all other cases, the parent node's branching rule is maintained as the default choice.

Experimentally, DASH compares well against static and randomly switching heuristics counterparts. The authors present a pair of variants (DASH+ and DASH+filt), allowing the choice of not switching heuristic inside a defined cluster, and performing a feature selection operation as well. The higher the degree of flexibility and information selection of the

algorithm, the better the numerical results seem to be.

Some supplementary remarks. In [36] as in [79], another characteristic of an ideal branching heuristic emerges. Namely,

(*iv*) given the highly dynamic and sequential nature of B&B, a branching scheme should be *adaptive*, not only with respect to different instances, but with respect to the whole tree evolution as well.

We observe that [36] constitutes an initial attempt of using the idea of exploiting a larger and more complete set of problems information by typical ML tasks such as clustering and dimensionality reduction. However, the method faces some limitations. First, the offline and rigid clustering upfront seems to clash with the dynamic and ever changing nature of B&B subproblems. Note that the identified need of considering the evolution of the tree, as expressed in (*iv*), needs to be balanced with (*iii*) time-efficiency, resulting in the depth and intervals prescriptions. Finally, the role played by different portions of the dataset within the various learning steps is not clear. A precise procedure of model selection should be performed, in order to avoid hidden overfitting and other pathological behaviors.

Before we move on to the next section, we summarize the recognized properties a learned branching rule ought to incorporate:

(*i*) node-efficiency,

(*ii*) focus on top-levels of the tree,

(*iii*) time-efficiency,

(*iv*) adaptiveness within the tree evolution.

We are going to see how similar considerations are addressed in the following few works, exploiting more closely the ML framework.

3.4.2   Novel ML-based branching heuristics

With the exception of DASH [36], all other works that were discussed up to now aim at defining a new branching strategy by means of using various types of information and originally assemble branching decision rules. We will present in this section some attempts in taking further the "precursors" underlying ideas, having as goal that of building learned branching

schemes. The novelty of these approaches resides in their more methodological use of the ML framework, as we presented it in Section 3.2.

Reliability branching and information-based branching [77] inspired the work of [38]. Given the fact that SB-like decisions are considered good decisions for branching, in that they minimize the number of explored nodes, but coming with a very high computational cost, the idea of approximating and speeding up SB has been already explored by methods such as RB and the *non-chimerical branching* (NCB, in short) [87]. In [38], the aim is that of learning one efficient approximation of SB by means of supervised learning techniques. The proposed method consists of two main phases. First, features are extracted to depict the state of a candidate branching variable within a specific node of the tree. The full SB decisions are recorded by solving to optimality a set of training instances, and a regressor is learned to predict estimated SB scores. After that, the learned heuristic is employed as branching heuristic for future B&B runs.

Before going further into details, it is worth to remark few things. Note that in the first part of the approach SB is still (heavily) employed, before the switch to the learned heuristic takes place. This reminds of procedures such as *hybrid strong/pseudo-cost branching* (SB+PC, in short) where SB is employed only until a certain depth, and then switched for PC [70], and reliability branching.[3] Indeed, the use of supervised learning calls for labels, meaning that one still needs good SB scores as examples from which to learn. Moreover, B&B trees are now fully explored in the training phase: the exploration is deeper than those proposed in [78] and [77], which focus instead on the top tree levels only. However, with respect to these two works, the expensive training phase is now performed once and for all. The offline upfront aims in fact at generalizing a prediction across all possible future instances, and hence does not need to be repeated for each of them.

Within the features design process of [38], the trade-off between relevance and expense is treated with care. Moreover, the authors identify other three desirable properties that a set of features for branching should include:

- *size-independence*, if one aims at learning a function able to generalize across instances of various size;

- *invariance* with respect to irrelevant changes within the instance, e.g., row or column permutations;

---

[3]Note that SB+PC and RB differ on the switch from SB to PC: at a certain fixed depth of the tree for the former method, whereas depending on each variable's reliability (i.e., past usage) for the latter.

- *scale-independence*, meaning that features should not change if parameters c, A and b (as in (3.1)) are multiplied by some factor.

The defined features are divided into three main groups.

- *Static problem features* describe the role of candidate variable x_i with respect to the problem's parameters c, A and b, and are computed only once.

- *Dynamic problem features* outline the state of variable x_i with respect to the current node LP solution.

- *Dynamic optimization features* represent the overall statistical effect of variable x_i with respect to the optimization process.

At a given node of the B&B tree, the features vector ϕ_i represents the state of the candidate variable x_i. The SB score y_i, explicitly computed for the set of training instances, completes the pair (features, label). The learning task is a regression one, and it is performed with *Extremely Randomized Trees* [88], also known as *ExtraTrees*. ExtraTrees is an averaging ensemble method, based on Decision Trees, which can be employed for classification as well as regression. The purpose of averaging is that of reducing the variance of a prediction by combining many predictors. In particular, ExtraTrees randomly perturbs the construction of the regression trees, selecting a random subset of features and drawing a random pool of thresholds to determine best splits.

A set of random, small-size problems is used for collecting SB-like examples, and the final training dataset includes 10^5 observations, i.e., pairs of features of a candidate variable at a given node and its SB score. A subset of instances coming from MIPLIB [81, 89] is employed for assessing the learned heuristic by comparing it with five concurrent ones: random branching, most-infeasible branching, non-chimerical branching, full strong branching, and reliability branching. Experiments are performed with and without time and nodes limits.

In general, results are inferior to the state-of-the-art RB, but still showing that the learned branching efficiently imitates FSB. As the authors point out, the reduced time spent at each node allows the learned scheme to explore more nodes, which is obviously a key of success. Slightly better results can be obtained when tuning the learning to specific classes of problems, suggesting one of many possible research directions. Further details about the ML-based approximation of SB can be found in [75].

Two are the main differences between [38] and [90], the authors' other attempt in developing a learned branching strategy. The explored paradigm is that of *online* learning. In contrast

with the offline upfront previously discussed, data is now generated and learned on-the-fly, within the B&B process itself. This implies that no preliminary and separated training phase is needed anymore. Still aiming at learning a fast approximation of SB, and keeping the same features of [38], the other novelty consists in the introduction of a reliability mechanism, very similar to the one in [70]. More specifically, depending on the number of times a real SB score was already computed for a certain variable (the RB threshold $\eta = 8$ is used), one could deem the candidate reliable, and hence trust an approximate version of its SB score. Otherwise, if the variable is deemed unreliable (i.e., the information exploited about it is not enough to portray it correctly) a real SB score is computed. At a given node of the B&B tree, every time a variable is not deemed reliable, the features vector is computed together with the SB score, and a new example (features, label) can be added to the training set. The learning is performed with a simple linear regression, guided by a line search gradient descent algorithm (see, e.g., [91]).

The defined *online learning* branching (olb) strategy exhibits at least one limitation, i.e., that of not adapting over time, with respect to possible changes of the variables dynamics along the B&B tree, as suggested by (*iv*). Indeed, at some point in the tree all variables would be deemed reliable, and the learning would stop updating. To fix the issue, the authors propose a *perpetual* version of olb (oplb). In short, the improved method allows the addition of new examples also when a variable is deemed reliable. Although the SB scores are not computed in the first place, features are stored for a reliable candidate at a given node; eventually, when both child nodes are explored, the SB information becomes readily available and a new pair (features, label) can be added to the dataset.

Both olb and oplb are compared on MIPLIB [81,89] against three other heuristics: full strong branching, reliability branching and the learned branching of [38] (actually, the one of [92], the preliminary version of the same paper). Results are interesting, in that both olb and oplb are competitive with RB in terms of nodes and time performance profiles. Again, methods and results are further discussed in [75].

The same goal of learning an effective approximation of SB is pursued in [37]. The authors aim at defining a method imitating SB in its node-efficiency, while being low-cost in terms of computational time and adaptive with respect to different instances to be solved. The scheme consists of three phases. First of all, during a *data collection* phase, SB is employed as variable branching rule up to a limited number of nodes $\theta(= 500)$ of the B&B tree. The performed SB decisions are observed and registered as (features, label) pairs in the training dataset. Second, a *supervised ranking* task is executed, and a ranking function learned. Finally, the *ML-based B&B* takes over: the optimization continues employing the learned

ranking function as branching heuristic, while SB is turned off.

The introduced ranking framework seems a natural approach for variable selection: predicting a ranking rather than a scalar score (as it is done by means of regression in [38] and [90]) is what a branching heuristic is ultimately pursuing. Note that the strategy resembles SB+PC and RB, in that SB is used only up to a certain point, i.e., while candidate variables are uninitialized or deemed unreliable. The outlined technique acts on-the-fly, without any expensive upfront, but does not adapt overtime, in contrast with the online perpetual approach (oplb) proposed in [90]. In this sense, all procedures (*learned* [38], *olb* [90] and *SB+ML* [37]) seem to suffer the same limitation to adapt with respect to the tree evolution. The perpetual version *oplb* [90] is the only method taking explicitly care of the adaptive issue. However, a little bit unexpectedly, it does not lead to significant improvements when compared to its halting counterpart.

Going back to [37], features are divided into two categories:

- *Atomic features* describe the role of a candidate branching variable within a particular node of the tree. In particular, 72 atomic measures are designed (in a fashion similar to [38]), and are further split into *static* and *dynamic.* The former set includes those characteristics of the problem shared by the whole tree (they are computed at the root node), the latter encompasses the traits associated with a particular LP node.

- *Interaction features* consist of products of two static features. The whole features vector can be interpreted as a degree-2 polynomial kernel $K(\mathbf{u}, \mathbf{v}) = (\mathbf{u}^T \mathbf{v} + 1)^2$, acting in the 72-dimensional space of atomic features. More details on kernel mappings can be found in [20].

Given the goal of learning a ranking function, instead of a regression one, while SB scores are computed for the training set of examples, they are not directly employed as labels. The proposed scheme is a binary labeling: labels are either 1 or 0, depending on their being or not in a fraction of top scoring variables at a given node. More specifically, denoting by $\mathcal{C}_j$ the set of candidate branching variables at node N_j, the best SB score is $SB_*^j = \max_{i \in \mathcal{C}_j}\{SB_i^j\}$; a label for variable x_i at node N_j is computed as

$$y_i^j = \begin{cases} 1, & \text{if } SB_i^j \geq (1 - \alpha) \cdot SB_*^j \\ 0, & \text{otherwise,} \end{cases} \tag{3.6}$$

where $\alpha \in [0, 1]$ decides the portion of variables that are considered good, i.e., sufficiently close to the maximum score. This relaxed definition of "best" branching variables allows

one to take into account many good candidates in the learning. Moreover, as the authors point out, this scheme should prevent the execution of irrelevant learning, such as the correct relative ranking of variables with low SB scores.

The ranking formulation follows a *pairwise approach*: pairs of candidates are considered, and the objective is to rank them as SB does. Formally, a set of pairs $\mathcal{P}_j = \{(x_i, x_k) : i, k \in \mathcal{C}_j$ and $y_i^j > y_k^j\}$ is considered for every node N_j, and the learned ranking seeks to violate as few as possible *pairwise ordering* constraints of type

$$\forall j \in \{1, \ldots, \theta\}, \forall (x_i, x_k) \in \mathcal{P}_j : y_i^j > y_k^j. \tag{3.7}$$

The Support Vector Machine (SVM) classification approach of [93], SVM^{rank}, optimizes an upper bound on the number of violated constraints in (3.7), and it is used to approximate the ranking problem. The learned ranking function is then directly plugged in the branching system.

The ranking heuristic SB+ML is compared against other four strategies: CPLEX default of version 12.6.1 (in the spirit of hybrid branching, [73]), SB, PC and SB+PC. The comparison shows that SB+ML solves more instances than both PC and SB+PC, requiring fewer nodes. These savings counterbalance the more time spent per node of SB+ML, which could be imputable to features computation.

A step further is taken in [94], who briefly explores the possibility of employing online and reinforcement learning in order to build a branching heuristic. The motivation of such an approach comes from the nature of B&B itself, which makes it possible to model the branching decision as a *multi-armed bandit* (MAB) problem [95]. In a MAB problem, at each round an agent selects one of many available actions (arms) and registers the reward associated with the performed choice. The intuitive goal is to identify and follow a sequence of actions maximizing the long-term reward (or minimizing the regret). Note that the B&B system can easily be interpreted in MAB terms: every node corresponds to a round, and every candidate variable to an available arm. A (not fully discussed) performance measure can then be used as reward function to guide the agent in its selections.

A preliminary setup mentioned in [94] outperforms on average PC. Although at its start, this seems a promising path to be explored.

Exploiting the framework of reinforcement learning (cf. Section 3.2.3) for variable branching, and more generally within the B&B technology appears suitable, given the inherent sequential nature of B&B. In particular, the idea that at a given node one should take future steps into account is already expressed in [76]. In that work, SB is interpreted as a greedy heuristic,

optimizing the dual-bound improvement (a reward) as much as possible at the current node only. Though the two-steps lookahead strategy devised is costly, we should maybe take into consideration the sequentiality of the whole tree-process in future research.

We summarize the three main ML-based contributions discussed in this section in Table 3.1. For each work we report the chosen learning setting, details on the employed test setting (dataset, solver's specifications, compared algorithms and measures) and a brief descriptive summary of the results.[4]

Table 3.1 Synoptic comparison of the three discussed ML-based branching heuristics. For each work we report: learning setting, test set composition and specifications, employed solver and tested settings, list of compared algorithms (novel methods are in bold), measures of comparison, and a descriptive summary of the results.

	M. Alvarez et al (2017)	M. Alvarez et al (2016)	Khalil et al (2016)
Learning setting	ExtraTrees for regression (offline, supervised learning)	Linear regression (online and adaptive supervised learning)	SVM^{rank} (learning-to-rank with pairwise approach, on-the-fly supervised learning)
Test instances	. 150 random . 44 MIPLIB 3.0 + 2003 small to medium size	. 44 MIPLIB 3.0 + 2003 10 seeds small to medium size	. 84 MIPLIB 2010 10 seeds
Solver	CPLEX 12.2	CPLEX 12.6	CPLEX 12.6.1
Setting(s)	w/ and w/o: heuristics, cuts, presolve, timelimit 10min, nodelimit 10^5	disabled presolve, timelimit 2h	cutoff provided, timelimit 5h, disabled heuristics, cuts at root only
Algorithms	. random branching . MIB . NCB . FSB . RB . **learned**	. FSB . RB . learned . **olb** . **oplb**	. CPLEX default . SB . PC . SB+PC . **SB+ML**
Measures	. closed gap (within limits) . # solved (within limits) . # nodes . time	. performance profiles (nodes and time)	. # unsolved . # nodes . time
Results summary	**learned** well imitates FSB. LP gap is improved w.r.t. FSB in the setup w/ limits, but RB dominates. Good also w/ cuts and heuristics in the setup w/o limits.	**olb** and **oplb** are competitive with RB in both performance profiles. The adaptive **oplb** does not significantly improve **olb**.	**SB+ML** solves more instances than PC and SB+PC. On average, it requires fewer nodes than PC and SB+PC, and slightly more than CPLEX default.

3.5 Searching the branching tree

As discussed, the search process of B&B highly relies on how the exploration of the decision tree is performed. The mechanism of implicit enumeration has two general goals:

(I) quickly find a good (possibly optimal) *integer feasible* solution;

(II) provide a *certificate of optimality* for the current incumbent, i.e., prove that no better solution exists.

Indeed, a solution of good quality helps the implicit enumeration as it allows one to discard useless branches, limiting the number of explored nodes and focusing only on worthy subproblems. Thus, it appears clear that the order in which the nodes of the branching tree are selected for exploration has a significant impact on the efficiency of the B&B method.

Many heuristic rules for searching the B&B tree have been proposed in the years, with the aim of attaining one goal or another, or trying to balance both objectives in some way. We will briefly discuss some of them in what follows. What is certain is that no single heuristic dominates the others, their performance very likely depending on the class of considered MILP problems.

Far from being confined to MILP research, the exploration of a decision tree is actually a very interdisciplinary theme, common in the AI community as well. As a consequence, one should not be too surprised to find out that the recent works applying ML techniques to the B&B tree search come from AI researchers. Similarly to how we discussed branching in Section 3.4, we will start in Section 3.5.1 by retrieving early optimization works somehow anticipating the development of adaptive and informed search heuristics. We will then present in Section 3.5.2 two recent attempts in this direction, employing different learning frameworks.

3.5.1 Preliminary considerations on search

We begin our agenda with the work of [15], which stands as a survey of B&B search heuristics as well as a baseline ground for a general discussion on the topic. In particular, the authors compare 13 different node selection methods, identifying three leading evaluation measures to rank them, in line with goals (I) and (II) above. The ranking is performed with respect to (in order of importance):

- value of the best solution obtained,

- provable optimality gap, and

- computing time.

The various heuristics are categorized into four different classes. Among *static methods*, depth-first and best-first searching paradigms are interpreted as extreme points of view on the node selection task: the former is associated with goal (I), the latter is linked to (II). This understanding naturally motivates *two-phase methods*, alternating depth-first and best-first in order to balance the search objectives. Further, the idea of making more intelligent selections in order to find new improved incumbent solutions is at the root of *estimate-based methods*. Criteria such as best-projection and best-estimate exploit the notion of estimating the value of the best feasible solution with respect to a certain subtree. Finally, *backtracking methods* also use estimates to guide the search, with the aim of avoiding superfluous nodes. As expected, such estimation rules become more realistic as they get more complex, involving many different characteristics (or features) such as pseudocosts, fractionalities and probabilities of successful rounding.

As the authors point out, given that the effectiveness of the above methods depends on the problem type, one would seek a search strategy that could adapt to different instances. In particular, some grade of adaptiveness could be pursued in the combination of best-first and depth-first strategies, which, borrowing a reinforcement learning vocabulary (see Section 3.2.3), corresponds to balancing exploration and exploitation in the B&B environment. Overall, it seems that the systematic analysis performed (for the first, and still most complete, time) in [15] can now be revisited in the light of modern ML techniques.

A different observation on the nature of the B&B tree underpins the work of [18]. The authors support the claim that high-sensitivity and erraticism are inherent properties of tree search, due to the very same exponential nature of the enumeration tree, which ought to be exploited in a beneficial way. Their *bet-and-run* approach first triggers randomization, producing few short runs. Among those different runs, a bet is made on the most favorable one, which is then alone brought to completion.

In more details, the algorithm makes use of a restart policy reminiscent of [77] and [78]. Within a *sampling phase*, $C(=5)$ random clones of the problem are created and solved up to $N(=5)$ nodes only. The best clone with respect to some aggregated indicator is selected and fully solved in the *long run*. Two are the key aspects of the method.

1. First, one needs to generate meaningful diversity while randomizing, without degrading the average performance. Moreover, the randomization should happen after the preprocessing and the solution of the root node, in order to limit the computational overhead.

The implemented strategy consists of temporary replacing the objective function of a clone with a random one, having fixed all nonbasic variables with nonzero reduced cost at their value in the root node optimal LP solution. Reoptimizing will lead to a different basis on the optimal face of the LP relaxation of the initial MILP, from which to start the search. A cap on the number of performed simplex pivots is enforced in order to maintain a short computing time.

2. Second, a selection rule for the "best" clone run must be defined, and hence one needs to identify some measures evaluating the performed (short) searches. As the authors recognize, erraticism itself precludes the definition of a perfect criterion. Hence, the aim is that of establishing a positive correlation between the clone to be selected and the *a posteriori* better run. Note that this notion naturally calls for a supervised learning framework, the *a posteriori* best run consisting in an example's label.

   The authors identify eight indicators of performance, with priority order, extracting information about: open nodes, lower bound improvements, infeasibilities and number of simplex iterations. The proposed evaluation scheme discards the indicators that do not provide discriminant information, and breaks ties to favor the first (default and unperturbed) clone. Note that the defined criterion bases its decision-making on the very beginning of the search, only.

The bet-and-run algorithm is compared with the default CPLEX setting (no dynamic search) and two *a posteriori* oracle algorithms, on 344 instances from MIPLIB 2010 [16] and COR@L [96]. The algorithm produces savings in terms of time and nodes for medium to hard instances, but the computational overhead does not payoff for easy instances. A modified version, denoted as *hybrid*, is tested, which prescribes to run CPLEX default for NR nodes to understand whether the instance at hand is "easy". If yes, the problem is solved by CPLEX with no modifications. Otherwise, perform the sampling phase of bet-and-run: if the selected clone is the unperturbed one, continue with default; else, continue with bet-and-run selection for the long run. The hybrid variation is beneficial on average, although it does not seem to fully solve the overhead issue.

Interestingly, throughout the paper, the authors themselves point out some possibilities for improvement that could involve the use of ML tools.

- Learning algorithms could provide a more sophisticated decision rule for the best clone selection criterion. A classification mechanism could improve the selection.

- While $NR = 500$ and 1000 are tested, the computation of the parameter could be

adaptive and performed on-the-fly, i.e., estimating the hardness of the remaining tree or, equivalently, the hardness of instance.

- The restart strategy could be refined by leveraging information of past runs.

We are now going to see how the main axes of these analyses on B&B search are reinterpreted in a learning framework.

3.5.2 Learning approaches to B&B search

Keeping in mind the recognized needs and goals of a search strategy for B&B, we will present in this section two very recent approaches for the topic. The tools employed in those attempts may appear uncommon to a MILP practitioner, and we will try to outline their potentials as well as their limitations.

In [41] the exploited framework is that of reinforcement learning (see Section 3.2.3). A multi-armed bandit (MAB, see Section 3.2.3) structure is proposed for MILP search, in the form of a modified version of the Upper Confidence bounds for Trees (UCT) [97] technique. Namely, UCT is a method for Monte Carlo Tree Search balancing exploration and exploitation, and it is based on the selection strategy of Upper Confidence Bounds (UCB1), which was introduced in [98]. In a nutshell, UCT works on an underlying tree T and consists of two alternating phases. Within the *node selection phase*, T is traversed from its root to a leaf node: at each node N the rule is to move to the child N' with higher *UCT score*; ties are arbitrarily broken. Once a leaf L is reached, a *tree update phase* is performed: an updated score is computed for L and it is propagated upwards to the root, following the outbound path in reverse order and adjusting the estimates for the encountered nodes.

The authors employ a simplified version of the UCT score of a node N (the $\varepsilon-$greedy version in [98]), consisting of a balanced sum of two terms:

$$\text{score}(N) = \text{estimate}(N) + \Gamma \cdot \frac{\text{visits}(P)/100}{\text{visits}(N)}, \tag{3.8}$$

where $\text{estimate}(N)$ is some measure of quality of node N, P is the parent node of N and $\text{visits}(\cdot)$ counts the number of times a node has been already visited by the search algorithm. The parameter Γ controls the balance between exploration (second term) and exploitation (first term): nodes with high estimate are pursued, but other nodes get priority if they have been visited only a fraction of the times compared to their siblings. In the original context of UCT (adversary game tree search), the $\text{estimate}(\cdot)$ values are initialized by *random playouts*:

the game is played many times until its very end by selecting casual moves, each play yielding a certain result that is used as measure of quality for the traveled path. Moreover, the tree update phase is carried out by a so-called *backup operator*, which usually assigns to a node N in the path from leaf to root the average of the values seen in the subtree rooted at N.

Given the differences between the original context of UCT and that of MILP, the algorithm must be appropriately modified. The goal of such adjusted UCT is that of guiding B&B search by expanding open nodes as UCT would expand them.

For a start, B&B tree search is a single-agent process, and it is clear that the MILP framework cannot afford random playout samplings for initialization purposes. Instead, the fact that branching provides guaranteed LP bounds is exploited, so that the quality estimates consist of normalized LP objective values. Having guaranteed bounds requires changes be made for the backup operator as well: instead of an averaging one, a *max-style* updating rule seems more suited. In short, each node's estimate is updated with the maximum between the estimates of its children nodes, so that at any N, estimate(N) equals the best objective value seen in the subtree rooted at N. Note that when a node is closed by B&B (i.e., fathomed), the search will not have any reason to visit it again. In this sense, exploitation is not always meaningful in this setting, and subtrees with no open nodes left should be disregarded by future UCT searches.

The UCT-based search strategy is compared with three others, namely, best-first, (graph-theory) breadth-first and CPLEX default heuristic. A general MILP solver performs B&B by internally maintaining the list of open nodes, but in order to apply the UCT-based method one needs to maintain an underlying entire tree structure to guide the search, i.e., nodes already explored are not removed. This additional architecture introduces significant overhead. To limit it, and supporting the notion that decisions are more crucial at the top levels of the tree, each tested strategy is performed on the first 128 nodes only, and then switched to the CPLEX default one. A total of 170 instances from MIPLIB 2010 [16] are used, with 600 seconds time limit, and the balancing parameter is tuned to $\Gamma = 0.7$. The geometric means of runtime, number of searched nodes and of simplex iterations are all improved by the UCT-based technique.

Some remarks before moving on. Note that the essential part of the reward measure consists of LP objective values, which is combined with the visits counter. Considering this, it seems that UCT improvements are gained thanks to a balanced usage of best-first and breadth-search-like schemes, without the exploitation of other state information. Moreover, the original UCT algorithm as in [97] treats every internal node as a different MAB problem, a property which the authors do not consider sustainable in the MILP context. However, detaching for a

moment from architectural issues, one could easily imagine bandit problems at every node of the B&B tree, eventually dealing with branching decisions, as proposed in [94]. An interesting question is whether the two processes of variable and node selection could be unified under the MAB scheme, or, more generally, within a (reinforcement?) learning framework.

Learn an adaptive and problem-specific search strategy is the goal of [42]. This work makes use of *imitation learning* (or *behavioral cloning*, see [99]), a paradigm very common in robotics. In general, imitative methods involve an expert performing some task, and having its actions recorded together with a description of the current situation. A dataset of (situation, action) pairs is given as input to a learning program, which then produces as output a set of rules (i.e., policies) reproducing the expert's behavior with respect to the performed task.

In the context of B&B, the expert part is played by a MILP solver and of a simple defined *oracle*. While it would be ideal to have an oracle expanding an optimal sequence of nodes and fulfilling (I) and (II) (and hence also (i), i.e., minimizing the number of explored nodes), the designed oracle only cares of (I). Namely, the explicitly declared goal is that of finding good (possibly optimal) solutions quickly, but without providing a certificate of optimality. This modeling choice is motivated by the wish of allowing a more aggressive pruning of the tree branches, and motivated by the idea that it may be possible in the future to reach a "user-specified trade-off between solution quality and searching time". Indeed, it is worth observing that the proposed framework does not guarantee an optimality certificate because it prunes subtrees potentially containing the optimal solution (besides the training phase in which the optimal solution is known).

More in details, the assumption is that optimal solutions of training problems are given. The oracle node selection rule π_S^* will always pursue the branches containing the optimal solution. Nodes expanded in this process are called *optimal nodes*; the non-optimal nodes are pruned from the tree by the oracle pruning rule π_P^*. The B&B is framed as a sequential process within the state space $\mathcal{S}$ consisting of the visited nodes and their LP bounds. Two policies that should guide the search are learned as rules. Namely,

- The (learned) *node selection policy* π_S prescribes which node should be expanded next. Namely, π_S provides a priority order for the queue of open nodes, deciding which one should be popped. The action space of π_S is {*select node N_i : $N_i \in$ queue of open nodes*}.

- The (learned) *node pruning policy* π_P determines whether a popped node is worth to be expanded. Note that in this context the terms "pruning" and "fathoming" are not interchangeable: a node that is not fathomed by infeasibility, bound or integrality could

still be pruned because of its non-optimality. The action space of π_P is $\{prune,\ expand\}$, so that π_P actually behaves like a binary classifier.

In fact, not only π_P, but the whole imitation problem can be reduced to supervised learning, as shown in [100]. The oracle actions a_t^* can be interpreted as predictions with respect to a features vector description of state s_t. Instead of forecasting a regression score, π_S itself can be framed as a classifier of pairs, for the problem where one aims at learning a ranking of the open nodes in queue. The procedure is in the spirit of that employed in [37] for variable selection purposes.

Features are divided into three categories.

- *Node features* include bounds and objective function estimation at a given node, together with indications about the current depth and the (parental) relationship with respect to the last processed node.

- *Branching features* describe the variable whose branching led to a given node, in terms of pseudocosts, variable's value modifications and bound improvement.

- *Tree features* consider measures such as the number of solutions found, global bounds and gap, with respect to the computed B&B tree as a whole.

Two different feature maps are defined for the policies: while π_S bases its predictions mainly on node and branching features, π_P employs mostly branching and tree features. All features are combined with the depth of the measured node by means of partitioning the tree in 10 uniform levels, and are appropriately normalized with respect to the root node's values. The designed characteristics do not constitute a computational burden, being easily obtainable from a MILP solver.

In the experiments, instances are borrowed from four diverse libraries. The MILP solver SCIP [31] is run to optimality and the delivered optimal solution is used within the oracle to initialize the training phase. The training of the policies is performed iteratively on problem classes. Every iteration provides updated information and collects new examples during the B&B runs, taking care of correctly ranking a node when it enters the open nodes queue and pruning it if non-optimal, after it is extracted from the queue. As already said, the learning of π_S and π_P is in fact a classification task, and attention is paid to properly tune parameters.

At test phase, the resulting algorithm, called *DAgger*, is compared with SCIP and Gurobi [101], taking into account the trade-off between runtime and solution quality in the comparison. The adaptive solver performs well on different classes of problems, and it seems to fulfill

the three-level adaptiveness sought by the authors, with respect to: (1) problem type, (2) specific instance and (3) different stages of the B&B optimization.

Further analysis shows the pruning policy having more impact, possibly due to the fact that other heuristic components interfere with node selection. Not so surprising are the findings of the features analysis. In general terms, π_S imitates depth-first search at the top of the tree, but also considers historical estimates in lower levels. The branching variable's measures seem to affect π_P, which keeps the pruning cautious when only few solutions are known.

Note that the branching decision is explicitly taken into account in the design of the search strategy by the second group of features, reflecting the idea that these two heuristic processes should be intertwined within the learning as they indeed are in the optimization. Moreover, the authors claim that the design of DAgger takes into account the complex sequential nature of B&B by modeling the influence of actions over future states, a task that could not be performed by standard supervised learning. However, the work is based on the assumption that current solvers' strategies are the "experts" to be imitated, and drops the seek of certified optimality for speed.

We summarize the two main ML-based contributions discussed in this section in Table 3.2. Again, for each work we report the chosen learning setting, details on the employed test setting (dataset, solver's specifications, compared algorithms and measures) and a brief descriptive summary of the results.

3.6 Overview and conclusions

In this paper, we have surveyed learning techniques to deal with the two most crucial decisions in the branch-and-bound algorithm for MILP, namely variable and node selections. Because of the lack of deep mathematical understanding on those decisions, the classical and vast literature in the field is inherently based on computational studies and heuristic, often problem-specific, strategies. Although our survey is mostly concerned with the recent methods that explicitly consider (machine) learning techniques, we have taken the perspective of interpreting some of the previous fundamental contributions in the light of those techniques, so as to give a more complete overview and to possibly outline new points of view.

It is worth observing that we have not touched in our discussion a nowadays fundamental component of branch-and-bound algorithms and codes for MILP, namely parallelization. Modern MILP solvers are developed, tested and used within multi-thread computing environments and more and more research is devoted to improve in the use of multi-threading. Although the papers we surveyed almost never discuss the issue, it is not difficult to imagine the use

Table 3.2 Synoptic comparison of the two discussed learning approaches to B&B search. For each work we report: learning setting, test set composition and specifications, employed solver and tested settings, list of compared algorithms (novel methods are in bold), measures of comparison, and a descriptive summary of the results.

	Sabharwal et al (2012)	**He et al (2014)**
Learning setting	UCT (reinforcement learning)	Imitation learning
Test instances	. 179 various benchmarks	. 36 MIK . 120 Regions . 40 Hybrid . 300 CORLAT
Solver(s)	CPLEX 12.3	. SCIP 3.1.0 (CPLEX 12.6 for LP) . Gurobi 5.6.2
Setting(s)	node and branch callbacks on, 600s timelimit	average runtime and # of nodes of the proposed B&B are used as timelimit for SCIP and nodelimit for Gurobi, respectively
Algorithms	. **UCT** . CPLEX default . best-first . breadth-first	. $\pi_\mathbf{S} + \pi_\mathbf{P}$ (selection + pruning) . $\pi_\mathbf{P}$ (pruning policy only) . SCIP (time) . Gurobi (node)
Measures	. runtime . # nodes . # simplex iterations	. speedup w.r.t. SCIP default . optimality gap . integrality gap
Results summary	**UCT**-based technique improves the geometric means of all considered measures.	Good adaptive performance on all classes of problems: $\pi_\mathbf{P}$ seems to have more impact, $\pi_\mathbf{S}$ likely interferes with other solver's components.

of the learning algorithms for both variable and node selections in a parallel environment. However, evil is in the details and the discussed paradigms have to be treated/extended with care.

Of course, variable and node selections are not the only important decisions in enumerative algorithms in general. One of the areas in which modern learning techniques could result crucial is that of predicting the difficulty of an instance, for example by taking the size and the shape of the enumeration tree into account. The problem of hardness prediction is not new. Since the first estimation of efficiency for backtracking methods [47], the question has been a common interest of the optimization and the ML communities, which developed their own algorithms in the past decades. Indeed, the practical impact of such a prediction is wide and important, especially given the time and resources limits that one has to confront when dealing with hard problems. Discussing this topic in details is outside of the scope of the present paper. However, again following the pattern of interpreting some old(er) contributions

in the light of modern learning algorithms and then considering the most recent works, we refer the interested reader to [48] for the former and to [35] and [49] for the latter.

Acknowledgments

CHAPTER 4 ARTICLE 2 – A CLASSIFIER TO DECIDE ON THE LINEARIZATION OF MIXED-INTEGER QUADRATIC PROBLEMS IN CPLEX

Authors: Pierre Bonami, Andrea Lodi and Giulia Zarpellon[1]
Submitted for journal publication. The work is an extension of a conference paper published in *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, CPAIOR 2018[2], which is included in Appendix A.

Abstract We translate the algorithmic question of whether to linearize convex Mixed-Integer Quadratic Programming problems (MIQPs) into a classification task, and use machine learning (ML) techniques to tackle it. We represent MIQPs and the linearization decision by careful target and feature engineering. Computational experiments and evaluation metrics are designed to further incorporate the optimization knowledge in the learning pipeline. As a practical result, a classifier deciding on MIQP linearization is successfully deployed in CPLEX 12.10.0: to the best of our knowledge, we establish the first example of an end-to-end integration of ML into a commercial optimization solver, and ultimately contribute a general-purpose methodology for combining learned predictions and Mixed-Integer Programming technology.

4.1 Introduction

While mathematical optimization intrinsically lies at the core of machine learning (ML) methods, recent years have seen a rise in the application of learned approaches to discrete optimization settings [6]. In particular, a successful paradigm has been identified in using ML within Mixed-Integer Programming (MIP) algorithmic frameworks as a way of complementing the capabilities of a solver and providing indications about structural decisions for which we lack in-depth understanding.

We position our work in this recent yet very fruitful research area, and consider Mixed-Integer Quadratic Programming problems (MIQPs). Despite the fact that modern MIQP solvers – and among those IBM ILOG CPLEX [28], our solver of choice – have been able to solve MIQPs for several years (see, e.g., [102]), the theoretical and computational implications of the employed resolution techniques are not fully grasped yet. We are interested in

---

[1]Authors are listed alphabetically, as is standard practice in Operations Research journals and conferences.
[2]Available at [26]. A preprint of this chapter is instead available at [27].

understanding whether to linearize the quadratic part of a convex MIQP, a decision that substantially conditions the downstream resolution algorithms operated by a solver. Currently, CPLEX users can utilize a switch parameter to specify whether a linearization step should take place during preprocessing, but it is not clear when this switch should be turned on or off in order to benefit the resolution process: when one considers a wide variety of problems the decision about whether to linearize is not clear cut.

Our goal in this paper is to use ML statistical tools to decide whether to linearize a convex MIQP or not. We make the empirical conjecture that some of the reasons leading to an algorithmic discrimination between the linearization approach (L, in short) and the non-linearization one (NL) might be linked to the formulation characteristics and the early stages of the optimization of a MIQP problem, and could hence be detected by a learning algorithm if enough relevant information was provided as input. The idea that perhaps MIQPs should be solved in a more flexible and adapted way was first suggested in [103], and naturally calls for a predictive machinery. In this sense, the question *linearize vs. not linearize* qualifies as a good quest for ML techniques, and it is naturally framed in a classification setting. We began to explore such classification approach in [26]: the developed framework took care of building a synthetic dataset, designing features and labels, and conducting preliminary learning experiments. We also defined new metrics to assess the quality of the prediction from the optimization standpoint, i.e., in terms of runtimes. Results were satisfactory, but limited by the fact that only artificial MIQPs were used. Moreover, the offline learning phase was not integrated in the solver.

4.1.1 Contributing a methodology

In the present work, we extend what was done in [26] along different directions, aiming at a tighter combination of the learning and the optimization perspectives. We resume from what we identified in [26] as future research plans and

- enlarge our dataset to include non-synthetic benchmark instances: we add to our pool of problems MIQPs from NEOS [104–106] submissions and CPLEX internal testbed;

- extend the feature design and feature selection process to achieve more detailed representations of MIQPs, of which we carry out a careful analysis;

- perform new learning experiments and explore different ways to incorporate optimization knowledge in the learning pipeline.

Finally, we implement our predictive framework in the solver ecosystem: as a practical out-

come, a learned classifier is deployed in CPLEX 12.10.0. These contributions allow us to establish the first example of an end-to-end integration of ML tools into a leading commercial solver – from the definition of an appropriate ML task responding to the algorithmic question of whether to linearize a MIQP, until the final deployment of the obtained prediction function within a complex solver environment. We believe that the methodology we designed over time, starting from the early attempt [26], could be applied to a variety of other heuristic tasks in the solver, and will serve as a reference in an area that is rapidly evolving and gaining attention. In this sense, the present work ultimately contributes a methodological process for the combination of ML and MIP technology: we share the questions that guided us in the development, the decisions and turns we had to take and the motivations behind them.

The paper content is shaped upon our methodological steps, which are outlined in Figure 4.1. We start by examining the MIQP algorithmic framework of CPLEX (Section 4.2), in order to identify and properly delimit our learning question (e.g., in terms of which MIQPs and algorithms are involved). The next step is building a dataset (Section 4.3). Targets capture the essence of a learning question, so their definition is of utmost importance, and we discuss two valid labeling procedures for discriminating between the L and NL methods. We approach feature design with questions like: what factors could be important for our decision? which traits of MIQPs might play a role in the algorithms we are trying to compare?, and address the need to gather (and generate) MIQP instances for data collection.

We follow ML best practices when defining learning experiments (Section 4.4), but we complement them with context-specific measures to evaluate the classifier performance in the solver. In fact, standard ML indicators cannot provide information on the impact of misclassification in terms of the metric that we use to compute targets. Baseline results serve us to verify the soundness of our approach and get an idea about the importance of the represented features, but the initial framework needs to be adjusted to be embedded in the solver. We ask ourselves what is ultimately viable and what changes are necessary to incorporate predictions in CPLEX: answers to such questions lead us to a substantial feature selection phase. To further condition predictions towards our true performance goal, we introduce domain-specific priors in the learning phase, and eventually recover information that was previously sacrificed. The experimental phase proceeds far from linearly, and we iterate step III (Figure 4.1) to attain satisfactory results. Finally, we discuss in Section 4.5 the practical implementation of a predictor in the CPLEX optimization pipeline, the required fine-tuning and the achieved outcome, before some concluding remarks in Section 4.6.

I.	**Understanding the algorithmic framework**	
	· Identify and delimit the learning question	
II.	**Building a dataset**	
	· Target definition via labeling procedure	
	· Feature design to represent the decision	
	· Gathering data: instances and analysis	
III.	**Learning experiments**	
	– Best practices and context-relevant evaluation metrics	
	· Baseline results	
	– Assessing feature importance	
	· Framework revision: necessary adjustments, feature selection	
	· Adding domain-specific priors to the learning phase	
IV.	**Implementing predictions in the solver ecosystem**	
	– Fine-tuned workflow	

Figure 4.1 ML in MIP technology: methodological steps.

4.2 The MIQP algorithmic framework in CPLEX

We consider Mixed-Integer Quadratic Programming problems, i.e., optimization problems in which a quadratic objective function is minimized over a set of linear constraints, and (a share of) bounded variables are required to be integral. We write a MIQP as

$$\min \left\{ \frac{1}{2} x^T Q x + c^T x : Ax = b, \ l \le x \le u, \ x_j \in \mathbb{Z} \ \forall j \in I \right\}, \tag{4.1}$$

where the matrix $Q = \{q_{ij}\}_{i,j=1,\dots,n} \in \mathbb{R}^{n \times n}$ defines the objective function together with $c \in \mathbb{R}^n$, while $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$ formulate linear constraints. Variables $x \in \mathbb{R}^n$ are bounded and $I \subseteq \{1, \dots, n\}$ denotes the set of indices of variables that are constrained to be integer. Without loss of generality Q is assumed to be symmetric. It is well known that (4.1) is $\mathcal{NP}$-hard (e.g., Max-Cut can be cast as a MIQP with binary variables).

When integrality requirements are dropped from (4.1), one obtains the (continuous) Quadratic Programming (QP) relaxation of the problem. If the matrix Q is positive semi-definite ($Q \succeq 0$), the quadratic form to minimize is convex and the corresponding QP can be solved in polynomial time; in this case, the QP relaxation is thus called *convex*. In the present work, we restrict ourselves to MIQPs whose Q matrix is positive semi-definite or can be made positive semi-definite by simple transformations, i.e., we only consider MIQPs that are usually regarded as "convex" by state-of-the-art MIQP solvers. Two simple transformations that can be applied to repair the indefiniteness of Q are the following.

Linearization of products involving binary variables Consider a binary variable x_j of (4.1), i.e., $l_j = 0$, $u_j = 1$ and $j \in I$. The product of x_j with any other bounded variable x_i satisfies the following linear inequalities [107]:

$$\max \left\{ \begin{array}{c} u_i x_j + x_i - u_i \\ l_i x_j \end{array} \right\} \leq x_i x_j \leq \min \left\{ \begin{array}{c} l_i x_j + x_i - l_i \\ u_i x_j \end{array} \right\}. \tag{4.2}$$

Whenever $x_j \in \{0, 1\}$ the inequalities of (4.2) turn into equations. Therefore, every such product $x_i x_j$ can be expressed using a new variable y_{ij} and the respective linear inequalities (4.2), with the corresponding entry q_{ij} then set to 0. Note also that $x_j^2 = x_j$ if $x_j \in \{0, 1\}$, so that all squares involving a binary variable can be moved from Q to the linear part of the objective as well.

Perturbation of the diagonal of Q for binary variables Again using the fact that for a binary variable $x_j^2 = x_j$, one has $x^T Q x + \sum_{j \in B} \rho_j \left(x_j^2 - x_j \right) = x^T Q x$, where $B \subseteq I$ denotes binary variables. The principal minor of Q corresponding to variables in B can thus be made positive semi-definite. In particular, if all non-zero products in Q involve at least one binary variable, Q can always be perturbed so that the resulting QP relaxation is convex. Note that the choice of an appropriate ρ is a non-trivial step. A simple way to ensure that the perturbed quadratic form $x^T Q x + \sum_{j \in B} \rho_j x_j^2$ has no negative eigenvalue is to directly use Q eigenvalues, though more advanced techniques leverage semi-definite programming and the linear constraints of (4.1) to produce tight QP relaxations [108, 109].

Leading solvers for MIQPs can perform either of the two operations above (linearize or perturb) at the beginning of the optimization, in a preprocessing phase. If the resulting problem has a convex QP relaxation it will be solved as a *convex MIQP*; otherwise, solving the QP relaxation itself is an $\mathcal{NP}$-hard problem [110] and more involved techniques are required. In this paper we only consider the former case. The state of the art for solving convex MIQPs usually employs computationally efficient algorithms for solving QP relaxations; in CPLEX, a simplex-based algorithm is preferred for its good restart properties. For the rest, the technology is similar to the one used for Mixed-Integer Linear Programs: a branch-and-bound tree search, augmented with cutting planes techniques, and heuristic procedures to obtain good feasible solutions [2]. Note that both approaches can also be applied when initially $Q \succeq 0$ – and in practice they are. In particular, the linearization step could reformulate an already convex MIQP into a Mixed-Integer Linear Program, thus deciding which resolution method and technology are applied to solve the problem.

4.2.1 The linearization option

Linearizing a convex MIQP has its benefits and inconveniences. The operation presents a technological advantage in that state-of-the-art solvers (CPLEX in particular) are typically better at solving Mixed-Integer Linear Programs than MIQPs: cutting plane techniques are more complete, and the algorithmic framework is overall more mature. However, the linearization step requires adding potentially many variables and constraints (depending on the non-zeros of Q), and the resulting Linear Programming (LP) relaxation may be very large and significantly slower. On the other hand, choosing to *not* linearize does not require additional variables, but one might be left to deal with a weaker QP relaxation (i.e., one with a worse bound), potentially due to a perturbation of Q diagonal to establish positive semi-definiteness.

As reported in [102], the linearization approach does not dominate in theory the non-linearization one. In CPLEX internal experiments, though, linearizing appeared to be superior on average to tackle convex MIQPs, and became the default method. Since version 12.6.0, CPLEX provides to users the possibility to switch the linearization mechanism on or off through the preprocessing parameter `qtolin`[3], whose automatic value corresponds to always linearize. But the linearization option is not always beneficial, as was reported in [103]. The following example shows that a range of situations can occur.

Example 4.2.1. We generate MIQP instances (see Section 4.3) of varying size and structural properties, and run CPLEX with both `qtolin` on and off with five different random seeds and a time limit of 2h. The following table reports for five problems the shifted geometric means of runtimes for the `L` and `NL` strategies, together with the problems' number of variables and constraints (n, m), the density of the Q matrix, and the percentage of "hard" eigenvalues of Q (i.e., those making the starting Q indefinite).

Table 4.1 Structural parameters and average runtimes for five synthetic MIQP instances.

	n	m	Density of Q	Hard Eigen.	L Time	NL Time
A.	150	5	0.20	0.00	**7.24**	7200.00
B.	175	1	0.57	0.00	1159.69	**251.34**
C.	100	11	0.96	0.32	**372.75**	819.26
D.	150	5	0.70	0.01	140.84	**136.48**
E.	125	10	0.95	0.51	7200.00	**1812.76**

Clearly, the initial convexity of Q itself does not decide which method between `L` and `NL` is the best suited to solve a MIQP, and Q density does not define the best option either.

---

[3]`https://www.ibm.com/support/knowledgecenter/SSSA5P_12.9.0/ilog.odms.cplex.help/CPLEX/Parameters/topics/QToLin.html`

Possibly, a combination of many factors together could parameterize the best solving mode. Note that choosing the correct strategy for problems A and E appears critical, in the sense that a wrong decision could result in the problem not being solved within the time limit. In contrast, for problems B and C the performance gap of L and NL is less pronounced, while for D the two methods are practically equivalent.

The main question addressed in this paper is to decide whether in the preprocessing phase one should linearize products involving binary variables, when solving convex MIQPs.

4.3   Building a dataset

To obtain predictions on MIQPs, we need to build a set of data-points, each representing a MIQP instance like (4.1) and the best decision for it between L and NL. More formally, we need to build a dataset $\mathcal{D} = \{(\mathbf{x}^k, y^k)\}_{k=1\ldots N}$: for every k, a vector of features $\mathbf{x}^k \in \mathbb{R}^d$ describes MIQP k, while a categorical label (target) y^k encodes the linearization decision. We explain in this section the target definition and the design of MIQP features, before discussing the dataset composition.

4.3.1   Labeling procedure

Given our question L vs. NL, we need to provide for each MIQP the answer corresponding to the better performing approach. We identify three possible scenarios, and therefore assign one among three categorical labels: L (*linearize*, i.e., qtolin on), NL (*not linearize*, i.e., qtolin off), and T (*tie*), when L and NL methods are comparable in terms of performance. Tracking tie cases provides a way of distinguishing between critical and non-critical problems, and can be helpful when evaluating the learned predictions.

To deal with the solver's performance variability [17], each instance is run in both qtolin modes with five different random seeds. We enforce a timelimit of 2h for each run, and collect data on final upper and lower bounds, resolution times and solver's solution statuses. We implement two checks to remove troublesome runs:

- Consistency check: on each seed, we compare best primal and dual bounds achieved by methods L and NL; when an inconsistency is found, the run on that seed is discarded;

- Solvability check: the run on a seed is discarded if neither L nor NL were able to solve the problem to optimality.

These checks on seed runs are reported in Algorithm 1, for the minimization case. After removing faulty runs and missing values from the data, we can decree the winner between L and NL or assign a tie T, for each seed. Algorithm 2 details the following $\mathsf{MultiLabel}$ labeling procedure. When both modes are able to solve the instance, running times are compared and a "seed win" is assigned if one method performed at least 10% better than the other one, opting for a tie alternatively. Instead, if only one method could solve the problem, it is decreed the winner for that run. A final label for each MIQP is determined by cumulative wins: L or NL are assigned only if their seed wins are consistent through the available runs, while T is returned otherwise. Note that the $\mathsf{MultiLabel}$ algorithm is based on the standard procedure employed in MIP development to compare two methods and determine their relative wins/losses.

In addition, we define a binary labeling scheme $\mathsf{BinLabel}$, reported in Algorithm 3. Unlike the multi-class procedure, $\mathsf{BinLabel}$ does not take into account consistent seed wins. Instead, it directly compares the shifted geometric means of running times for L and NL, across the seeds passing the checks. Eventually, ties are broken using shifted geometric means of the number of nodes. The comparison of computing times in $\mathsf{BinLabel}$ does not use a 10% threshold, but the resulting scheme is nevertheless consistent with the $\mathsf{MultiLabel}$ one: L and NL labels assigned in the multi-class procedure remain the same in the binary one. In other words, one can interpret $\mathsf{BinLabel}$ as a way of turning T samples obtained with $\mathsf{MultiLabel}$ into L and NL cases.

Algorithm 1: Checks on MIQP runs

Input: For a minimization MIQP (4.1), lower bounds lb_*, upper bounds ub_* and statuses $status_*$ collected on runs in modes $* \in \{\mathsf{L}, \mathsf{NL}\}$, on a same seed s. Tolerance parameter $constol (= 1 \times 10^{-5})$.

Output: `True` if the run passes consistency and solvability checks, `False` otherwise.

1: $LB := \max\{lb_\mathsf{L}, lb_\mathsf{NL}\}$, $UB := \min\{ub_\mathsf{L}, ub_\mathsf{NL}\}$

    Consistency check

2: **if** $(LB - UB > constol \cdot \max\{|LB|, |UB|, 1\})$ **then**

3:     **return** False   ▷ *discard the run because inconsistency was found*

4: **end**

    Solvability check

5: $\mathtt{solved} = [\mathtt{optimal}, \mathtt{infeasible}]$ ▷ *statuses corresponding to solved runs*

6: **if** $(status_\mathsf{L} \notin \mathtt{solved})$ and $(status_\mathsf{NL} \notin \mathtt{solved})$ **then**

7:     **return** False   ▷ *discard the run because not solved within timelimit*

8: **end**

9: **return** True

Algorithm 2: Labeling procedure – MultiLabel case

Input: For a MIQP, times $t_{s,*}$ and statuses $status_{s,*}$ for all seeds s that passed checks and
 $* \in \{\mathsf{L}, \mathsf{NL}\}$. Parameters $p(= 0.1)$ to compare runtimes, $\Delta(= 3)$ to compare seed wins.
Output: A label in $\{\mathsf{L}, \mathsf{NL}, \mathsf{T}\}$.

Seed wins
1: $wins_{\mathsf{L}} := 0, wins_{\mathsf{NL}} := 0$
2: $\mathtt{solved} = [\mathtt{optimal}, \mathtt{infeasible}]$ ▷ *statuses corresponding to solved runs*
3: **for** seed s **in** passed seeds **do**
4:     **if** $(status_{s,\mathsf{L}} \in \mathtt{solved})$ and $(status_{s,\mathsf{NL}} \in \mathtt{solved})$ **then**
5:         **if** $t_{s,\mathsf{L}} < (1-p)t_{s,\mathsf{NL}}$ **then**   ▷ *L significantly better than NL on s*
6:             $wins_{\mathsf{L}} \leftarrow wins_{\mathsf{L}} + 1$
7:         **else if** $t_{s,\mathsf{NL}} < (1-p)t_{s,\mathsf{L}}$ **then**   ▷ *NL significantly better than L on s*
8:             $wins_{\mathsf{NL}} \leftarrow wins_{\mathsf{NL}} + 1$
9:         **end**
10:     **end**
11:     **if** $(status_{s,\mathsf{L}} \in \mathtt{solved})$ and $(status_{s,\mathsf{NL}} \notin \mathtt{solved})$ **then**
12:         $wins_{\mathsf{L}} \leftarrow wins_{\mathsf{L}} + 1$
13:     **else if** $(status_{s,\mathsf{L}} \notin \mathtt{solved})$ and $(status_{s,\mathsf{NL}} \in \mathtt{solved})$ **then**
14:         $wins_{\mathsf{NL}} \leftarrow wins_{\mathsf{NL}} + 1$
15:     **end**
16: **end**

Winner label assignment
17: **if** $wins_{\mathsf{L}} \geq wins_{\mathsf{NL}} + \Delta$ **then return** label L
18: **else if** $wins_{\mathsf{NL}} \geq wins_{\mathsf{L}} + \Delta$ **then return** label NL
19: **else return** label T
20: **end**

Algorithm 3: Labeling procedure – BinLabel case

Input: For a MIQP, times $t_{s,*}$ and nodes $nodes_{s,*}$ for all seeds s that passed checks and
 $* \in \{\mathsf{L}, \mathsf{NL}\}$. A function *sgmean* to compute shifted geometric means, with shift $\varepsilon(= 1)$.
Output: A label in $\{\mathsf{L}, \mathsf{NL}\}$ or $\mathtt{None}$.

1: $Time_* := sgmean(t_{s,*} : s$ passed checks) for $* \in \{\mathsf{L}, \mathsf{NL}\}$
2: $Nodes_* := sgmean(nodes_{s,*} : s$ passed checks) for $* \in \{\mathsf{L}, \mathsf{NL}\}$

Time and nodes comparison
3: **if** $Time_{\mathsf{L}} < Time_{\mathsf{NL}}$ **then return** label L
4: **else if** $Time_{\mathsf{NL}} < Time_{\mathsf{L}}$ **then return** label NL
5: **else**
6:     **if** $Nodes_{\mathsf{L}} < Nodes_{\mathsf{NL}}$ **then return** label L
7:     **else if** $Nodes_{\mathsf{NL}} < Nodes_{\mathsf{L}}$ **then return** label NL
8:     **else return** $\mathtt{None}$
9:     **end**
10: **end**

4.3.2 Feature design

A raw formulation like (4.1) cannot be fed directly as input to a learning algorithm, so we need to represent a MIQP via a vector of numerical features $\mathbf{x} \in \mathbb{R}^d$, which should condense what we suspect are the important pieces of information leading to an algorithmic discrimination between L and NL. We describe a MIQP instance in its mathematical, optimization and computational properties, by means of a set of 60 hand-crafted features. For feature design, we reinterpret few ideas from the recent works [37] and [35]. We mostly capture *static* information from the initial formulation, but given the impact of the linearization and perturbation steps on the quality of the root dual bound [108] (and hence on the success of the subsequent optimization process), we also extract data from the *preprocessing* phase and the resolution of the *root node relaxation*, for both L and NL. Features are defined in such a way to be comparable across a variety of instances, as they should express common characteristics of MIQPs. With respect to our early work [26], the feature set has been revised and extended, with the goal of better capturing the composition of matrix Q and the changes induced by preprocessing.

Static features Properties of a MIQP that can be read from the formulation (4.1) are basic information about the size of the problem (number of variables and constraints) and the proportions of variables of each type (binary, general integer and continuous). The composition of the symmetric matrix Q can be detailed by inspecting the presence of different types of non-zero bilinear products, in and out of the main diagonal, and in particular the appearance of non-linearizable terms. Additionally, we examine spectral properties of Q such as rank and proportions of zero and "hard" eigenvalues. Connectivity degrees of variables appearing in $x^T Q x$ are also tracked, and we compute proxies for potential increases in variables and constraints sizes after linearization. We record the composition and density of the linear term c of the objective function. For constraints, we inspect variables' involvement (per type) and the density of matrix A.

Preprocessing features After the preprocessing phases of L and NL, we record the actual increases in number of variables and constraints, relative to the original dimensions n and m. The density of the constraints matrix after preprocessing is also examined and compared with the one of A, and between the two methods.

Root node features To measure the performance difference between L and NL in solving their respective relaxations, we collect and compare runtimes and dual bounds achieved after

the root node resolutions.

In total, we collect 44 static, 11 preprocessing and 5 root node features; we report them in Table B.1 in Appendix B. At this stage, features are still extracted offline: we compute static traits with the CPLEX Python API after reading MIQP instances. Necessary information from preprocessing and the root node resolutions of L and NL are gathered during the runs of the labeling procedure (Section 4.3.1); dual bounds and runtimes at the root node are aggregated using arithmetic and shifted geometric means, respectively, across the runs that passed the checks of Algorithm 1.

4.3.3 Instances

We aim to compile a dataset of MIQPs that is heterogeneous and relevant for the L vs. NL question, and representative of the variety of cases that can occur, as we saw in Example 4.2.1. Driven by the need of more problems than what available libraries offer, we first create synthetic MIQP instances. The generation procedure takes into account different structural parameters (such as size, density and spectrum of Q) and multiple types of constraints to obtain heterogeneous MIQPs. We refer to the resulting dataset of 2640 MIQPs as setD, and to [26] for more details on data generation.

Examining the problems that were generated for [26], we observed the presence of instances with high density due to a dense encoding and near-to-zero coefficients q_{ij}. We hence decided to apply a numerical correction to the dataset, enforcing sparsity of Q matrices. On the one hand, such correction disrupted the spectral properties of some unstable instances, which could now be read by the solver as general nonconvex ones, and consequently rejected.[4] On the other hand, we deem the corrected instances to be more stable and meaningful than their original versions. Instances of setD have been contributed to the MINOA open-source benchmark library [111].

For this work, we enlarge our MIQP dataset to include non-synthetic benchmark instances from NEOS submissions and problems of CPLEX internal testbed:

- neos contains 945 MIQPs that were submitted to the NEOS server with CPLEX as the specified solver. Instances have been collected from submissions between April 2015 and January 2018 and cleaned for duplicates;

- miqp contains 522 problems that constitute the CPLEX internal MIQP dataset. Differently from setD, miqp is dominated by the presence of very structured combinatorial

---

[4]This behavior can be explained by the fact that we generated Q matrices with non-full rank, and that zero eigenvalues are not really null in floating point operations.

MIQPs, like Max-Cut and Quadratic Assignment Problems. Note that instances from the literature (e.g., QPLIB ones [24]) are also included in this set.

Table 4.2 Dataset composition in terms of labels, for both labeling schemes. Percentages refer to the corresponding row counts (#).

	MultiLabel				BinLabel		
	#	L (%)	T (%)	NL (%)	#	L (%)	NL (%)
`setD`	1821	614 (35.2)	841 (46.2)	339 (18.6)	1322	942 (71.3)	380 (28.7)
`neos`	480	49 (10.2)	426 (88.8)	5 (1.0)	137	93 (67.9)	44 (32.1)
`miqp`	284	101 (35.5)	149 (52.5)	34 (12.0)	191	133 (69.6)	58 (30.4)
Total	**2585**	791 (30.6)	1416 (54.8)	378 (14.6)	**1650**	1168 (70.8)	482 (29.2)

Altogether, `setD`, `neos` and `miqp` amount to 4107 MIQP instances, which we run in both `qtolin` modes, for five random seeds, on a cluster of identical 12 core Intel Xeon CPU E5430 machines running at 2.66 GHz and equipped with 24 GB of memory, with CPLEX version 12.8.0. After performing Consistency and Solvability checks, 2585 problems remain $(1821, 480, 284$ from `setD`, `neos` and `miqp`, respectively). We then compute labels with both schemes MultiLabel and BinLabel, and extract features as described in Section 4.3.2. The composition of the dataset in terms of labels is reported in Table 4.2; note that a proper BinLabel could not be assigned in 935 cases due to ties both in runtimes and number of nodes[5], so that only 1650 problems are available in the binary setting. While the MultiLabel scheme produces almost 55% of tie cases, BinLabel yields a 70-30% repartition between L and NL, respectively. As shown in Figure 4.2a and 4.2b, the proportion of problematic eigenvalues of matrix Q, as well as its density, span the entire [0,1] range in the full dataset.

4.4 Learning experiments

We perform learning experiments on the entire dataset with multi-class labels $\{L, NL, T\}$ given by MultiLabel (2585 points), as well as on the binary subset of samples with targets defined by the BinLabel procedure (1650 points).

As classification models, we test Logistic Regression (`LogReg`), Support Vector Machine (`SVM`) with RBF kernel [112], a single Decision Tree (`Tree`) and Random Forests (`RF`) [113]. We specify a grid of values for the main hyper-parameters of each model, in order to search the best combinations. All models can perform both multi-class and binary classification, and

---

[5]This happens because we set all runtimes smaller than or equal to 0.1 seconds to be precisely 0.1.
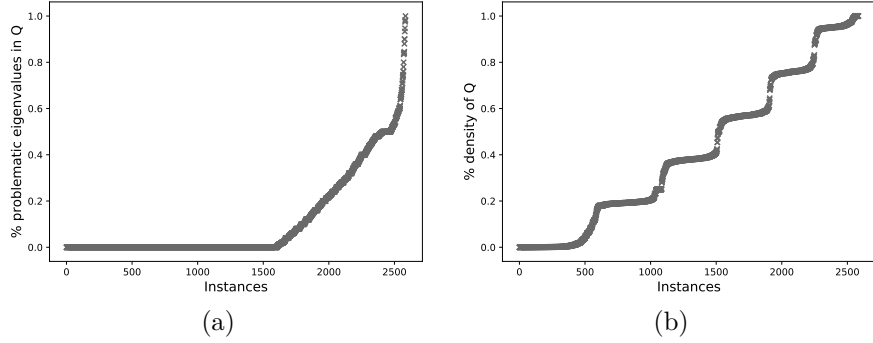
Figure 4.2 (a) Fraction of problematic (hard) eigenvalues of Q, and (b) density of Q in the full dataset (2585 instances).

are compared with a dummy classifier (`dum`) following a stratified strategy (i.e., generating predictions according to the class distribution of the training set).

For each of our learning experiments, we randomly split the available data into a training and a test sets using a 75-25% ratio. We perform a training phase with 5-fold cross validation to grid-search models' hyper-parameters, and a test phase on the neutral test set. When splitting the data and defining folds, general proportions of labels are maintained in each subset; features are standardized with respect to each subset, by removing the mean and scaling to unit variance. Each type of experiment is repeated for five different random seeds: within the learning pipeline, randomization mostly affects the determination of the train/test splits, but can also impact the definition of some predictive models (e.g., `RF`). Practically, learning experiments are implemented in Python 3.5 with Scikit-learn 0.20.0 [63], and run on a dual Intel(R) Xeon(R) Gold 6142 CPU @ 2.60GHz, equipped with 512GB of RAM.

Metrics Especially in our context, it is important to quantify the performance of the trained classifiers not solely with respect to standard classification measures: from an optimization standpoint, we need to determine how effective and valuable our learned approach proves to be when practically solving MIQPs, and compare it to the solver current strategy. For this reason, we rely on a set of heterogeneous metrics to assess the performance of the predictors.

For a vector of true labels y and a vector of *predicted* labels $\hat{y}$, we compute the accuracy of the predictions over a (test) set of size K as

$$\text{accuracy}(y, \hat{y}) = \frac{1}{K} \sum_{k=1}^{K} \mathbb{1}_{\{y^k = \hat{y}^k\}}. \tag{4.3}$$

We also report f1-scores, i.e., harmonic means of precision and recall for the predictions (see [20] for details). More generally, a weight w^k can be associated to each sample k, to get a *weighted* accuracy score

$$w\text{-accuracy}(y, \hat{y}, w) = \frac{1}{\sum_k w^k} \sum_{k=1}^{K} w^k \cdot \mathbb{1}_{\{y^k = \hat{y}^k\}}. \tag{4.4}$$

To compensate for (and get a sense of) the effects of class-imbalance in the data, one can evaluate a *balanced* accuracy score. By defining *class* weights w_{class} as the uniform weights over samples of the same class, one can compute

$$\text{b-accuracy}(y, \hat{y}) = w\text{-accuracy}(y, \hat{y}, w_{class}). \tag{4.5}$$

However, from the perspective of practically solving MIQPs, misclassifying critical problems has more severe effects than predicting the wrong method for an instance in which L and NL show instead comparable performances. As we saw in Example 4.2.1, misclassifications are neither all equally important nor bad. In order for our measurements to reflect the quality of the predictions from the solver's standpoint, we introduce a notion of sample weights linked to the runtimes of L and NL. A natural way of measuring how critical a MIQP problem is – i.e., how different the methods perform, and hence how important it is to classify the sample correctly – is that of considering the shifted geometric mean of the runtimes difference, as in

$$w_{time} := sgmean(|t_{s,L} - t_{s,NL}| : s \in \text{seeds}). \tag{4.6}$$

Weights w_{time} can be easily obtained from the benchmark runs of the labeling procedure. We then measure accuracy "with respect to times",

$$\text{t-accuracy}(y, \hat{y}) = w\text{-accuracy}(y, \hat{y}, w_{time}). \tag{4.7}$$

On a similar note, we compare prospective runtimes of predictors. For each classifier *clf*, we associate a vector of "predicted" times t_{clf} to the vector of its predicted labels $\hat{y}_{clf}$: for every (test) sample k, we select $Time_*^k$ for the corresponding predicted label $* \in \{L, NL\}$. As in Algorithm 3, $Time_*^k$ is defined as the shifted geometric mean of runtimes, across the available seeds from the labeling benchmark. If a tie T was predicted for sample k by *clf*, we set t_{clf}^k to be the average of $Time_L^k$ and $Time_{NL}^k$. Likewise, we compute t_{def} and t_{target} for the solver's default strategy (always linearize) and the ideal classifier that perfectly predicts the true targets, respectively. A simple sum of such predicted (prospective) runtimes enables one to get a sense of how effective a learned discrimination between L and NL can be. We

define

$$\sigma_{clf} := \sum_{k=1}^{K} t_{clf}^k, \tag{4.8}$$

and compare both $\sigma_{clf}/\sigma_{\mathtt{target}}$ (the smaller the better, ideally 1) and $\sigma_{clf}/\sigma_{\mathtt{def}}$. Note that $\sigma_{\mathtt{target}}/\sigma_{\mathtt{def}}$ naturally provides a bound to how much a classifier can improve on the current default solver setting, in a given subset of samples $k \in \{1, \ldots, K\}$.

4.4.1 Baseline results

As a baseline experiment, we train classifiers on the initial set of 60 features (`Initial`), in both multi-class and binary settings. Results reported in Table 4.3 are averages of the scores across five tries. With respect to traditional classification measures, all classifiers are exhibiting good performance, with `RF` and `SVM` usually being the best performing models. Scores accounting for runtimes in their definition (i.e., t-accuracy, $\sigma_{clf}/\sigma_{\mathtt{target}}$, and $\sigma_{clf}/\sigma_{\mathtt{def}}$) appear consistent with the classification ones. In particular, `RF` yields at least a 14% improvement on `def` in both settings: `def` uses 16% or more time than `RF` to solve test instances. Classification scores in the multi-class configuration are generally higher than those in the binary one. An inspection of the confusion matrices allows to assess that classifiers in the multi-class setting are in fact very good at correctly classifying T cases, with errors mostly happening when distinguishing between L and NL. Given that the BinLabel scheme in fact transforms some tie cases into L and NL samples, the classification of T which was so accurate in the MultiLabel setting translates into less clear-cut separation in the binary one. Nonetheless, despite misclassification happening more frequently in the BinLabel setting, t-accuracy and ratios of prospective runtimes remain high for `RF` and `SVM`. Again, misclassifications do not have all the same impact in terms of solver performance, and runtime-based metrics show that classifiers are still able to predict correctly on many critical binary samples. For these reasons, we decide to focus our subsequent experiments in the binary setting only.

Feature importance To get a sense of the importance of each feature in the prediction, we analyze the importances scores of the trained `RF` models. Such scores, computed by Scikit-learn, consist of nonnegative scalar values summing up to 1 (among all features), and represent the mean decrease in impurity [114] for each feature. Simply put, the predictive power of an attribute is quantified in terms of the depths in the decision tree at which the attribute is used to create a node split, following the rationale that attributes used at the top practically affect more samples. We average the features' scores across the five `RF` models trained in both data setting, and consider the 10 top-ranked features, which we report in Table B.2 in Appendix B. For MultiLabel, top features are mostly from preprocessing and

Table 4.3 Baseline results for the two data setting; 60 `Initial` features are used for learning. Reported values are averages across five experiments of the same type, on different seeds.

	dum	LogReg	RF	SVM	Tree	def	target
accuracy	0.41	0.85	0.90	0.87	0.87	-	-
b-accuracy	0.33	0.78	0.85	0.81	0.81	-	-
f1-score	0.41	0.85	0.90	0.87	0.87	-	-
t-accuracy	0.29	0.93	**0.96**	0.95	0.91	-	-
$\sigma_{clf}/\sigma_{\texttt{target}}$	3.06	1.28	**1.15**	1.17	1.39	1.33	1.00
$\sigma_{clf}/\sigma_{\texttt{def}}$	2.27	0.96	**0.86**	0.88	1.03	1.00	0.75

(a) MultiLabel, `Initial` features

	dum	LogReg	RF	SVM	Tree	def	target
accuracy	0.60	0.76	0.83	0.80	0.80	-	-
b-accuracy	0.50	0.69	0.77	0.73	0.77	-	-
f1-score	0.59	0.76	0.83	0.79	0.80	-	-
t-accuracy	0.69	0.91	**0.96**	0.95	0.89	-	-
$\sigma_{clf}/\sigma_{\texttt{target}}$	2.46	1.42	**1.17**	1.26	1.53	1.37	1.00
$\sigma_{clf}/\sigma_{\texttt{def}}$	1.78	1.04	**0.85**	0.92	1.11	1.00	0.73

(b) BinLabel, `Initial` features

the root node resolution; this suggests that T cases can be well detected thanks to this type of non-static information. In the binary configuration, instead, a common subset of features starts to emerge: together with few non-static features, attributes describing the composition of matrix Q appear. In particular, measures of density and of the presence of binary variables gain relevance. In Figure 4.3 we plot four relevant features across the multi-class dataset. The density of Q (Figure 4.3a) and the relative dual bound difference between L and NL (Figure 4.3d) clearly help in discriminating between classes. Together with the relative increase in number of variables after L preprocessing (Figure 4.3b), they also reflect the trade-off between size and strength of the (re-)formulation, which is crucial for a successful resolution of MIQPs [108] .

4.4.2 Feature selection

Up to now we performed data collection and learning experiments offline, tracing over the outline of what had been done in [26]. However, the ultimate goal of the present work is that of tightly integrating a predictive tool with the MIQP solver, a task that requires rethinking our initial framework and adjustments of various kind. Mainly, instead of data being collected once and for all, feature extraction will need to be performed online, when a
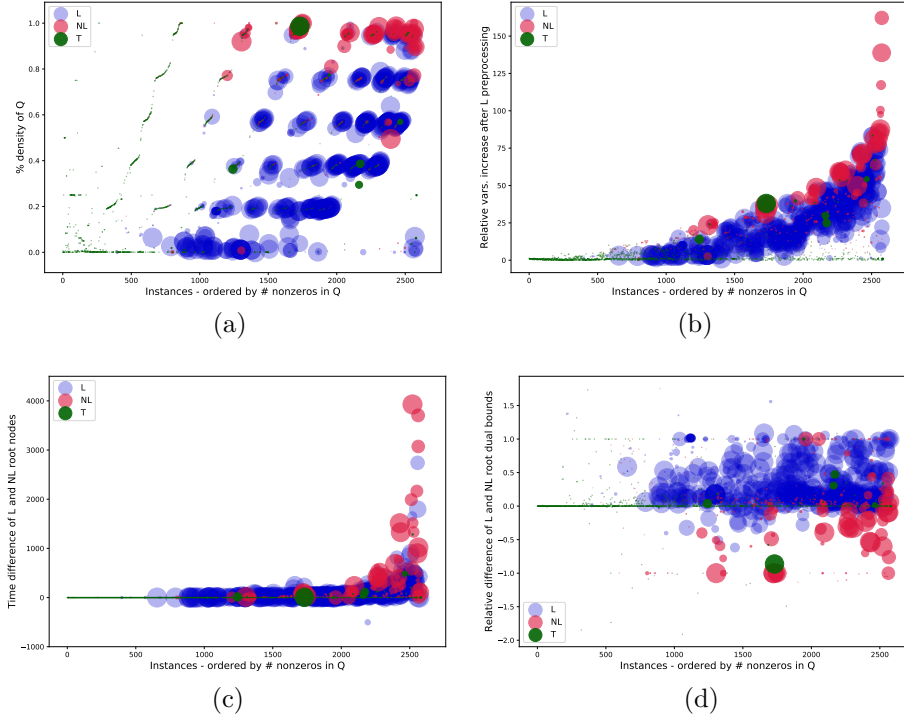
Figure 4.3 Four relevant features in the full dataset: (a) density of Q, (b) relative increase in number of variables after L preprocessing, (c) root node runtimes difference between L and NL, (d) root node bounds difference between L and NL. Colors match MultiLabel targets, while bubble size is proportional to weights w_{time}. In all plots, instances are ordered by the number of non-zeros in Q.

MIQP instance is presented to the solver. More generally, one has to understand when the prediction should take place, with respect to the resolution pipeline and the solver's various functionalities – a consideration that, in turn, affects which type of input can be available for the predictor model. Moreover, not all the hand-crafted features prove to be useful for good classification: in fact, the presence of irrelevant features in the input may induce over-fitting, besides entailing extra computational cost. One generally needs to compromise between the predictive power of some features and the possibility of efficiently computing them in the solver: attributes related to root node information are certainly useful for classification but expensive to get, as they would require to solve the root node twice. Some static features involving a spectral decomposition of Q are also not viable in an online procedure.

Practically, these considerations altogether motivate the revision of the hand-crafted feature set. We drop features that are not accessible for an online solver computation; in particular, we remove from the initial 60 traits:

- most of the features regarding the spectrum of the Q matrix, only keeping information

Table 4.4 Description of features in the `Selected` subset (21).

Name	Description
Static features	
RBin	Ratio of binary variables over n
RContInt	Ratio of continuous and integer variables over n
RNnzDiagContInt	Ratio of non-zero (nnz) coefficients in Q diagonal for continuous and integer variables
OutDiagDensity	Density of non-diagonal entries of Q
QDensity	Density of Q
RBinBin	Ratio of nnz products between binary variables in Q
RContContInt	Ratio of nnz products between continuous or integer variables in Q
RNonLinTerms	Ratio of nnz non-linearizable terms, over n^2
RelVarsLinInc	Relative size increase of potential linearization, over n
RLinSizes	Sizes m/n ratio after potential linearization
NormMaxDegBin	Maximum connectivity degree in Q among binary variables, over $n-1$
NormMaxDegContInt	Maximum connectivity degree in Q among continuous and integer variables, over $n-1$
RNnzContIntLin	Ratio nnz continuous and integers variables in linear term
ConssDensity	Density of constraints matrix A
RConssInt	Ratio of constraints involving integer variables, over m
RQRankEig	Rank of Q over n (i.e., ratio of nnz eigenvalues of Q)
HardEigenPerc	Portion of problematic (hard) eigenvalues in Q
Preprocessing features	
prep_RelVarsIncL	Relative variables increase after L preprocessing
prep_RelConssIncL	Relative constraints increase after L preprocessing
prep_RSizesL	Sizes m/n ratio after L preprocessing
prep_ConssDensityL	Density of constraints matrix after L preprocessing

on the proportion of hard eigenvalues and zero ones (i.e., measures relative to Q rank);

· features gathered from root nodes resolution for both L and NL, and those from the preprocessing step of NL;

· features prone to numerically ill behaviors (e.g., those involving comparisons of A, b and c coefficients), in order to avoid scaling issues in the learning phase and remove dependencies from each instance's parameters.

After these reductions, we end up with a set of 35 features. Note that the only non-static features kept are those extracted from the L preprocessing step, which is more expensive than the NL one, but appeared more useful for predictions in the baseline experiments. We then proceed with a phase of further feature selection. Feature selection is an inherently iterative phase in the learning pipeline, and we try various procedures – from filtering based on a single SVM model or Decision Tree, to cross-validating the best subset using ensemble methods. We also examine the performance of different feature subsets with respect to the

regression task that will be included in the learning pipeline (see Section 4.4.4). At the end, we decide to keep a subset of 21 features (17 static and 4 from `L` preprocessing) that are consistently identified throughout the experiments as relevant and leading to satisfactory classification performances. We describe this final subset of attributes in Table 4.4, and refer to it as `Selected`.

We run classification experiments in the binary setting; results are reported in Table 4.5. All scores appear decreased with respect to the baseline (cf. Table 4.3); in particular, performance in terms of prospective runtimes is generally reduced. Relevant features consolidate in the binary configuration, consistent with what already observed in the baseline experiments: with features describing the initial Q composition (particularly in terms of binary variables appearance), traits on the effects of linearization (e.g., with respect to problem size and A's density) are regularly in the top-10 (cf. Table B.2).

4.4.3 Using runtime weights

After a necessary reduction in input features, classification and runtime-related scores dropped considerably. In this and the next sections, we introduce some changes in the learning process to condition predictions towards our true performance goal and thus improve the optimization performance of the classifiers. Improving the predictions from the optimization point of view concretely means making the classifiers more attentive to critical data-points. A very natural idea to incorporate the knowledge of whether a sample is critical is to use w_{time} as defined in (4.6) as *sample weights*, i.e., to work with a weighted dataset. In SVM models, for example, the use of weights on the points has the effect of re-scaling the penalty parameter, so that during training the classifier will be emphasised to get high-weight points correctly. In a single Decision Tree, instead, weights would modify the classes' probabilities in the identified split regions of the feature space.

Another possible way of introducing a prior about critical instances is by defining a custom loss function designed to penalize misclassification proportionally with the criticality of samples. In this respect, we define

$$\texttt{WTarLoss}(y, \hat{y}) := \frac{1}{\sigma_{\texttt{target}}} \sum_{k=1}^{K} w_{time}^{k} \cdot \mathbb{1}_{\{y_k \neq \hat{y}_k\}}, \tag{4.9}$$

a weighted loss with respect to `target` runtimes. For a misclassified sample k ($y_k \neq \hat{y}_k$), the weight w_{time}^{k} is a proxy for the difference between the prospective runtime of the classifier and the `target` one, i.e., of $t_{clf}^{k} - t_{\texttt{target}}^{k}$. We try `WTarLoss` as scoring function for cross-validation, so that during training the best combinations of classifiers' hyper-parameters will be chosen

Table 4.5 Results for the binary setting using the 21 `Selected` features. Reported values are averages across five experiments of the same type, on different seeds.

	dum	LogReg	RF	SVM	Tree	def	target
accuracy	0.60	0.76	0.77	0.77	0.75	-	-
b-accuracy	0.50	0.67	0.70	0.69	0.70	-	-
f1-score	0.59	0.74	0.76	0.76	0.75	-	-
t-accuracy	0.69	0.92	**0.94**	0.92	0.90	-	-
$\sigma_{clf}/\sigma_{\text{target}}$	2.46	1.37	**1.32**	1.40	1.49	1.37	1.00
$\sigma_{clf}/\sigma_{\text{def}}$	1.78	1.00	**0.96**	1.01	1.07	1.00	0.73

based on this score (the lower the better).

Results in both the sample weights setting and with the use of `WTarLoss` are reported in Table 4.6. Overall, it appears clear that incorporating some prior knowledge on which instances are critical via weights substantially helps to improve the classifiers prospective optimization performance. Measures of t-accuracy and runtimes ratios strengthen in both setups, with sample weights especially boosting `SVM` (cf. Table 4.7a). Classification measures are comparable between settings, and with respect to previous results in which no weight information were used (cf. Table 4.5). Note that the use of sample weights leads to lower b-accuracy scores, i.e., there is a marked imbalance in terms of which class is correctly predicted. In this setup, confusion matrices reveal that `L` is the predicted label for a higher number of samples, and most misclassifications happen in the form of a `NL` wrongly predicted as `L`. This might be linked to the fact that values of w_{time} show different distributions when restricted to `L` and `NL` samples; Table 4.7 presents descriptive statistics for weights in `BinLabel` data. While $w_{time} \in [0, 7200)$ hits the same min and max values for both classes, mean and 75% percentile values indicate that weights are higher for `L` samples. In other words, there is more to lose (on average, in our data) when misclassifying `L` for `NL` than vice versa.

4.4.4 Regression of root bounds information

Among the factors affecting the final runtime of a MIQP, an important one clearly is the quality of the dual bound reached at the root node, i.e., the strength of the initial problem relaxation. As we already noticed, features comparing root node information of `L` and `NL` methods were deemed very useful for correct predictions (cf. Table B.2 and Figure 4.3d). We try to incorporate dual bounds information without solving the root node twice, by approximating a root feature via learned regression. In particular, we select `root_RelSignRDBDiff`, i.e., the relative signed difference of root dual bounds in `L` and `NL`, and use it as target to train a Support Vector Regression model (`SVR`) with a nonlinear RBF kernel. For the minimization

Table 4.6 Results for the binary setting using the `Selected` feature subset (21), with sample weights w_{time} and custom cross-validation scoring function `WTarLoss`. Reported values are averages across five experiments of the same type, on different seeds.

	dum	LogReg	RF	SVM	Tree	def	target
accuracy	0.60	0.77	0.75	0.76	0.75	-	-
b-accuracy	0.50	0.64	0.66	0.63	0.62	-	-
f1-score	0.59	0.74	0.74	0.73	0.72	-	-
t-accuracy	0.69	0.95	0.95	**0.96**	0.94	-	-
$\sigma_{clf}/\sigma_{\texttt{target}}$	2.46	1.25	1.24	**1.21**	1.30	1.37	1.00
$\sigma_{clf}/\sigma_{\texttt{def}}$	1.78	0.92	0.90	**0.88**	0.94	1.00	0.73

(a) `BinLabel`, `Selected` features, w_{time} as sample weights

	dum	LogReg	RF	SVM	Tree	def	target
accuracy	0.60	0.76	0.76	0.76	0.75	-	-
b-accuracy	0.50	0.67	0.68	0.67	0.69	-	-
f1-score	0.59	0.75	0.75	0.75	0.75	-	-
t-accuracy	0.69	0.93	**0.94**	0.93	0.92	-	-
$\sigma_{clf}/\sigma_{\texttt{target}}$	2.46	1.36	**1.28**	1.35	1.43	1.37	1.00
$\sigma_{clf}/\sigma_{\texttt{def}}$	1.78	0.99	**0.93**	0.98	1.02	1.00	0.73

(b) `BinLabel`, `Selected` features, `WTarLoss`

Table 4.7 Statistics for w_{time} with respect to classes, in the `BinLabel` data. We report: count, mean, standard deviation, min, max and percentiles values.

	#	mean	std	min	25%	50%	75%	max
L	1168	1938.44	2853.36	0.0	0.0	36.08	4366.82	7199.90
NL	482	432.63	1433.96	0.0	0.2	1.72	20.11	7199.90

case (dual bounds are lower bounds),

$$\texttt{root\_RelSignRDBDiff} = \frac{lb_{\textsf{L}} - lb_{\textsf{NL}}}{1e{-}10 + \max(|lb_{\textsf{L}}|, |lb_{\textsf{NL}}|)}, \tag{4.10}$$

the metric being positive when L's bound is better, negative otherwise. Recall that bounds $lb_{\textsf{L}}$ and $lb_{\textsf{NL}}$ are arithmetic means of those benchmarked during the labeling procedure. In fact, this feature partially aligns with the binary labeling: when linearizing is the best choice, that will be reflected in the bound quality 91% of the times; when NL is the target, instead, the NL bound is actually better than the L one only 23.6% of the times (cf. also Figure 4.3d).

Practically, we need to allocate part of our data to train the SVR. In our experimental pipeline we now first perform a 30-70% split for training and testing the regression; we restrict classi-

fication to the `SVR` test set only, i.e., we further divide the 70% portion into a 75-25% split for training and testing classification models. The predictions of the trained `SVR` are added to the 21 input features of the `Selected` subset, and used for classification. Note that the entire classification phase only relies on *predicted* values of `root_RelSignRDBDiff`, never true ones.

The average mean square error of `SVR` across five experiments is 0.1012. Classification-wise, results improve when the regressed root information is exploited. Table 4.8 reports scores for both kind of weights integration. If classification metrics are comparable with the previous cases, the use of `SVR` is particularly helpful to improve runtime-related scores. In the sample weight case (Table 4.9a) models improve their performance, the only exception being `SVM`. When `WTarLoss` is used instead (Table 4.9b), `SVM` is the best performing classifier in terms of prospective runtimes ratios, scoring a 16% improvement on `default` runtimes. Feature importance scores as assigned by `RF` models identify the same top-10 attributes in both setups, with the predicted version of `root_RelSignRDBDiff` ranking at positions 4 and 5, respectively.

4.5   Implementing predictions in CPLEX

For the actual implementation of our trained predictors into CPLEX, we select an `SVM` model trained in the setting with `SVR` regression performed on `Selected` features and `WTarLoss` used as custom scoring function (cf. Table 4.9b). Overall, both SVM and RF models performed consistently well in our multiple experiments, but we ultimately opt for a SVM model over a RF one because of its easy-to-implement decision functions, whose coefficients and support vectors can be directly extracted with Scikit-learn.

We devise the following workflow in the solver: a MIQP problem is read and `L` preprocessing performed; after features are internally computed, the predictive pipeline starts: the trained `SVR` predicts a proxy of the root feature `root_RelSignRDBDiff`, which is added as input for `SVM` classification. If `L` is the predicted label, then the optimization continues; otherwise, the original model is resumed and the `NL` preprocessing and optimization applied to it. We refine the process to take care of two special cases:

(i) the problem is already solved during linearization preprocessing; in such case, no prediction is needed;

(ii) it may happen that during `NL` preprocessing on the original problem the solver fails to establish convexity and rejects the instance; in this case, we disregard the classifier's prediction and forcibly fall back to the `L`-preprocessed problem, to continue with the `L` resolution process.

Table 4.8 Results for the binary setting using the `Selected` feature subset and predicted feature `root_RelSignRDBDiff` (21+1) from `SVR`, with sample weights w_{time} and custom cross-validation loss function `WTarLoss`. Reported values are averages across five experiments of the same type, on different seeds.

	dum	LogReg	RF	SVM	Tree	def	target
accuracy	0.60	0.75	0.74	0.74	0.73	-	-
b-accuracy	0.51	0.61	0.61	0.62	0.59	-	-
f1-score	0.60	0.71	0.71	0.71	0.69	-	-
t-accuracy	0.67	**0.97**	0.95	0.94	0.94	-	-
$\sigma_{clf}/\sigma_{\texttt{target}}$	2.59	**1.17**	1.26	1.33	1.27	1.42	1.00
$\sigma_{clf}/\sigma_{\texttt{def}}$	1.78	**0.82**	0.88	0.90	0.89	1.00	0.70

(a) `BinLabel`, `SVR` + `Selected` features, w_{time} as sample weights

	dum	LogReg	RF	SVM	Tree	def	target
accuracy	0.60	0.76	0.76	0.77	0.74	-	-
b-accuracy	0.51	0.65	0.66	0.66	0.64	-	-
f1-score	0.60	0.74	0.75	0.75	0.72	-	-
t-accuracy	0.67	0.95	0.95	**0.96**	0.91	-	-
$\sigma_{clf}/\sigma_{\texttt{target}}$	2.59	1.25	1.24	**1.19**	1.48	1.42	1.00
$\sigma_{clf}/\sigma_{\texttt{def}}$	1.78	0.88	0.87	**0.84**	1.02	1.00	0.70

(b) `BinLabel`, `SVR` + `Selected` features, `WTarLoss`

For CPLEX internal fine-tuning, we refine the dataset by removing 252 instances solved by L preprocessing (i.e., those for which we do not actually need to train a classifier) and adding 40 large ones ($n \geq 10,000$) for which spectral features could not be previously computed, but are now available via the internal solver implementation. The resulting dataset amounts to 1674 instances when BinLabel is performed. As before, there is a 70-30% proportion between L and NL classes. We select the final SVM model from a last round of training, and embed its decision function in the solver.

As a final experiment, we run both CPLEX versions 12.9.0 and 12.10.0, the latter incorporating the learned classifier, which is run by default, on the test set (on which the classifier was not trained) over five seeds. Note that the solving environment now presents some differences with respect to the setting in which we computed targets: for labeling, version 12.8.0 was used together with a time limit of 2 hours; now the time limit is raised to 10,000 seconds, and computations run on a more powerful cluster of identical machines with 16 core Intel X5650 processor at 2.67GHz, 24 GB RAM and using 12 threads. The classifier ultimately yields a 28% improvement of running time over the previous default strategy; the measure
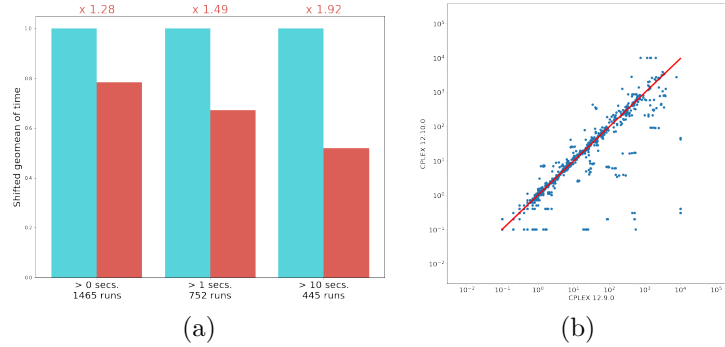
Figure 4.4 Comparison of MIQPs runtimes between CPLEX 12.9.0 and CPLEX 12.10.0 on the test set: (a) bar plot (CPLEX 12.10.0 in red), (b) scatter plot of running times in log scale.

increases to 92% when considering runs[6] taking more than 10 seconds to solve with either version (Figure 4.4a). Figure 4.4b reports a scatter plot of MIQPs running times between versions 12.9.0 and 12.10.0. While using the classifier results in slower runs for some models, the degradation is generally limited and compensated by improvements of several order of magnitudes. In particular, only two runs present a degradation of more than one order of magnitude (the worst case being 13 times slower), while 58 (resp. 30) of them show an improvement of more than one (resp. two) order of magnitude.

Future developments   As our metrics show, there surely is room for improving predictions. At the very least, the classifier could be periodically updated to incorporate newly available MIQP instances: most of the computational effort needed to maintain a growing dataset would be spent on label computation. Nonetheless, the question arises of how to compare prospective classifiers to the current one. In this respect, the definition of a fixed, shared MIQPs test set (as those available for other ML applications, e.g., for object recognition) could make future comparisons easier, but it definitely is a non-trivial task. Being this the first time a predictor is fully integrated in a MIP solver, we do not know all the answers upfront; we are curious to see how this classifier (and more generally this research field) will practically evolve.

4.6   Conclusions

We considered convex MIQPs and the question of whether to linearize the binary components of their quadratic objective in order to solve them. We translated the problem into

---

[6]As it was for labeling, a "run" corresponds to a MIQP model solved with a specific seed.

a classification task and addressed it with ML techniques. The developed framework aims at embedding a predictive function in CPLEX: with this goal in mind, we contributed a methodological process for combining ML and MIP technology, and thoroughly revised our initial work [26]. We built a dataset of synthetic and real-world instances, proposing labeling schemes and carefully engineering features to describe MIQPs and the decision of whether to linearize them. Learning experiments as well as evaluation metrics were designed to integrate the optimization knowledge in the learning pipeline. In particular, we experimented with runtime weights, a custom scoring function and with the regression of an attribute about the root node bounds. Finally, we carefully considered how to include a predictor in the solver ecosystem. As a result, a SVM classifier deciding on MIQP linearization is implemented in CPLEX 12.10.0, establishing to the best of our knowledge the first example of a learning-based tool deployed in a commercial optimization solver.

CHAPTER 5   ARTICLE 3 – LEARNING MILP RESOLUTION OUTCOMES BEFORE REACHING TIME-LIMIT

Authors: Martina Fischetti, Andrea Lodi and Giulia Zarpellon[1]

Published in *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, CPAIOR 2019.[2]

Abstract   The resolution of some Mixed-Integer Linear Programming (MILP) problems still presents challenges for state-of-the-art optimization solvers and may require hours of computations, so that a time-limit to the resolution process is typically provided by a user. Nevertheless, it could be useful to get a sense of the optimization trends after only a fraction of the specified total time has passed, and ideally be able to tailor the use of the remaining resolution time accordingly, in a more strategic and flexible way. Looking at the evolution of a partial branch-and-bound tree for a MILP instance, developed up to a certain fraction of the time-limit, we aim to predict whether the problem will be solved to proven optimality before timing out. We exploit machine learning tools, and summarize the development and progress of a MILP resolution process to cast a prediction within a classification framework. Experiments on benchmark instances show that a valuable statistical pattern can indeed be learned during MILP resolution, with key predictive features reflecting the know-how and experience of field's practitioners.

5.1   Introduction

Within the realm of discrete optimization, we consider Mixed-Integer Linear Programming (MILP) problems, of the form

$$\min\{c^T x : Ax \geq b, x \geq 0, \ x_i \in \mathbb{Z} \ \forall i \in \mathcal{I}\}, \tag{5.1}$$

where $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, $c, x \in \mathbb{R}^n$ and $\mathcal{I} \subseteq \{1, \dots, n\}$ is the set of indices of variables that are required to be integral. We do not assume A, b having any special structure (as it is, e.g., for Traveling Salesman Problem instances). Models like (5.1) can be used to mathematically describe a number of different real-world problems, and are daily deployed across a wide spectrum of applications – network, scheduling, planning and finance, just to mention a few.

Despite being $\mathcal{NP}$-hard problems, MILPs are nowadays solved in very reliable and effective

---

[1] Authors are listed alphabetically, as is standard practice in Operations Research journals and conferences.
[2] Available at [29].

ways, ultimately based on the divide-and-conquer paradigm of Branch and Bound (B&B) [3]. State-of-the-art optimization solvers, such as IBM-CPLEX [28], experienced a dramatic performance improvement over the past decades, due to both hardware and software advances (see, e.g., [1, 2]). Nonetheless, the resolution of some MILPs can prove to be challenging for solvers, and may require hours of computations, so that the experimental practice of imposing a time-limit (TL) to the MILP resolution process is not only very reasonable, but well established too. However, it would be useful to get a sense of the optimization trends after only a fraction of the specified TL has passed, and ideally be able to tailor the usage of the remaining resolution time in a more strategic and flexible way.

We aim to predict whether a generic MILP instance will be solved before timing out, only relying on information from a first portion of the resolution process. More specifically, given problem P and a time-limit TL, we look at the partial resolution of P, up to a certain time τ, $0 < \tau < TL$, and ask whether P will be solved to proven optimality within TL. We summarize the partial resolution of P, and exploit machine learning (ML) tools to cast a prediction about it being solved or not before TL. Thus, the prediction we aim at is one that takes as input (a summary of) the evolution of a partial MILP run, up to time τ, and outputs a yes/no response, in the framework of binary classification. Note the inherent difference between our approach and the problem of directly predicting the "difficulty" of a MILP instance – e.g., in terms of tree-size [48, 115] or runtime prediction, the latter being a common interest for both the optimization and the ML communities since the work of Knuth [47] (a more recent approach can be found in [35]).

The sequential nature of B&B makes it natural to interpret our question as a sequence classification task. However, the transformation of a stream of data from the MILP resolution process into a valid input for traditional classification algorithms cannot be performed with off-the-shelf techniques [116]. To this end, we design specific features to describe the development and behavior of a MILP run in a quantitative way, taking into account the complex interplay between the solver's components. The broad generality of the proposed features makes them apt to be re-used every time one needs to evaluate the B&B development of a general MILP, thus conferring even more impact to this contribution, especially given that applications of ML to discrete optimization have lately been flourishing as recently surveyed in [6]. For example, in the context of MILP, ML has been proposed to establish good solver's parametric configurations [34]; learn heuristics for B&B (see [25] for a survey); choose resolution options ([26, 43, 44]), and also predict solution-related outcomes ([54, 55]). Our work represents a novel contribution in this thread of research: ML is employed to provide an accurate prediction on the resolution outcome of MILPs, which can readily be implemented within solvers to enable tailored optimization and enhance the comprehension of the reso-

lution process, too often hard to unravel given the solver's complexity. In fact, despite the abundance of data and events in the MILP resolution framework, to the best of our knowledge no statistical analysis presently happens within the solver; in particular, information is not exploited in any structural way via ML algorithms to make decisions. Applying to generic MILP problems and opening new opportunities on the solvers' side, our results affect a broad audience and assume greater methodological relevance for the discrete optimization community.

We end the introduction by stressing that discovering early in the process that the run will very likely not terminate with a proof of optimality is of fundamental value for MILP development, and opens promising scenarios for both developers and end-users. Indeed, on the one hand, MILP developers can adapt the resolution through algorithmic changes in the attempt of avoiding the issue, or can switch mode so as to try to improve the incumbent solution as much as possible giving up optimality. On the other hand, this can be achieved by an end-user too, although that would likely require restarting the run with a different parameter setting. Finally, note that the indicators we developed could, in turn, shed some light on the type of required algorithmic changes.

5.2 Background: solving MILPs

As already mentioned, the resolution of MILPs is fundamentally based on the B&B paradigm. In its basic version, B&B sequentially partitions the solution space of (5.1) into sub-MILPs, which are mapped into nodes of a binary decision tree. At each node, the integrality requirements $x_i \in \mathbb{Z}$ for variables $i \in \mathcal{I}$ are dropped, and a *linear*, or *continuous, relaxation* (polynomially tractable) of the sub-problem is solved, providing a valid lower bound to the optimal solution value of the original MILP. When in the relaxed solution all variables $x_i, i \in \mathcal{I}$ take integer values, the solution is feasible for (5.1) as well and provides an upper bound of its optimal value. Otherwise, variables $x_i \notin \mathbb{Z}, i \in \mathcal{I}$ are *integer infeasible* (iinf) and among them one is selected for further branching: the tree is extended with two additional child nodes so that the current relaxed solution is removed from the sub-problems' feasible space; the new nodes also inherit from their parent an estimate of the objective function value. Global lower and upper bounds (called *best bound* and *incumbent*, respectively) are maintained throughout the resolution process and smartly used to prune unpromising regions of the feasible space, so that the resulting algorithm is only implicitly enumerating the exponentially many solutions of (5.1). The normalized difference between global bounds (known as *gap*) allows to measure at any point in time the quality of a solution and the progress of the optimization.

For example, CPLEX implements the following (relative) gap measure:

$$\text{gap} = \frac{|\text{best bound} - \text{incumbent}|}{1e^{-10} + |\text{incumbent}|}. \tag{5.2}$$

A MILP is solved when the gap is fully closed, i.e., when it reaches 0, with upper and lower bounds coinciding (up to numerical tolerances). The branching and bounding operations are combined with other solver's building blocks – the cutting planes algorithm [9], presolving, primal heuristics – to form a very rich and interconnected resolution framework [2], in which single events and data become hard to disentangle.

The ability to identify the resolution phases of a MILP [14] and analyze the outputs of the B&B algorithm can help recognizing causes of performance issues, and explaining instance-specific trends [117]. In particular, many indicators interact in describing the progress of the MILP resolution process, and need to be taken into account when casting a prediction about the resolution outcome. To provide a simple example, we plot in Figure 5.1 basic information from the resolution log of CPLEX, for an "Easy" instance of MIPLIB 2010 [16]. We report the development of the global bounds and the gap, the number of *nodes left* (i.e., the leaves yet to be explored) and the *depth* of the nodes as the algorithm traverses the tree. The interconnection between these figures is, for this easy case, quite clear to observe: for example, an update of the incumbent value naturally reduces the gap, triggers a drop in the number of nodes left (due to pruning by bound), and possibly ends a (depth-first) dive in the tree traversal exploration, a common practice when looking for initial feasible solutions with primal heuristics.

5.3   Problem formalization

We can re-phrase our question more formally by considering a MILP P, a time-limit TL, and a certain percentage ratio $\rho \in [0, 1]$ yielding $\tau = \rho \cdot TL \in [0, TL]$. We solve problem P with time-limit TL and take into account the evolution of its resolution process up to time τ. We denote with t_{sol}^P the moment in which P is solved to proven optimality by the solver. We want to describe and evaluate the progress (in other words, the "*work done*") in solving P, given that only a share of the total available time has passed; ultimately, we aim at casting a prediction on such a description. With respect to the defined parameters, we achieve 100% of work done at t_{sol}^P, and 100% of available time at TL. In practice, there is a discrepancy between t_{sol}^P and TL, the latter specified by a user, the former unknown and subject to variability.

Graphically, one could depict the advancement of the solver with a non-decreasing "*progress*
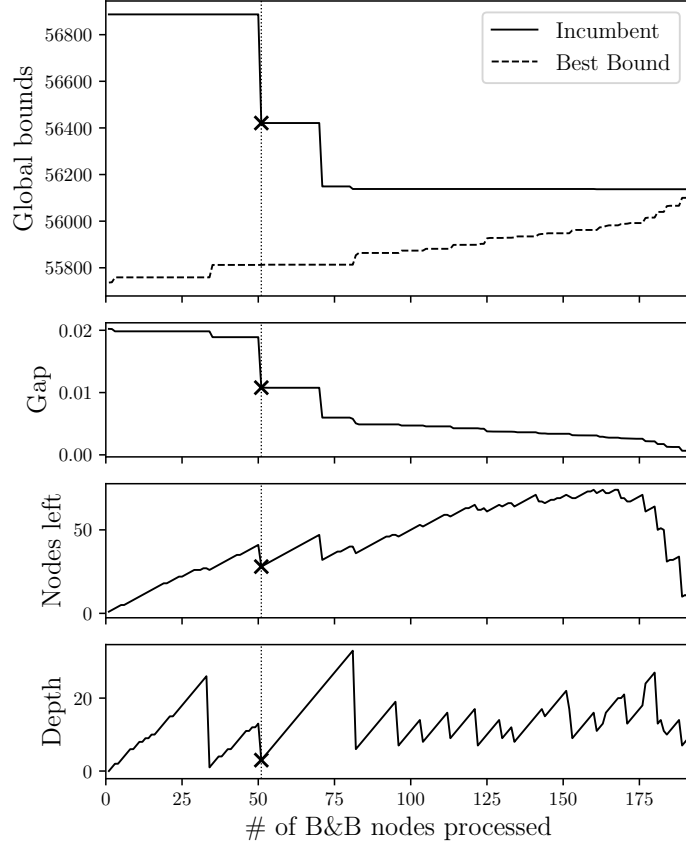
Figure 5.1 Basic information from the CPLEX log from the resolution of MIPLIB 2010 instance `air04`. Interpreting the evolution and interaction of these indicators enables a quantitative description of the optimization process.

measure", describing the proportion of work done given the proportion of time passed (Figure 5.2). Our classification question translates precisely into predicting whether the 100% of the work will be done before TL, i.e., whether $t_{sol}^P \leq TL$, only observing the resolution up to time τ. The function we aim to learn is thus the indicator function $\mathbf{1}_{\{t_{sol}^P \leq TL\}}$.

The task of feature design, on the other hand, aims at defining the progress measure used to represent the % of work done, given the triplet (TL, ρ, P). Instead of relying on a single feature to describe the optimization process (as could be done, e.g., using the gap), we try to capture the complexity of MILP resolution by considering heterogeneous measurements, and design a feature map Φ, describing the progress measure for (TL, ρ, P) with a vector in $\mathbb{R}^d$.
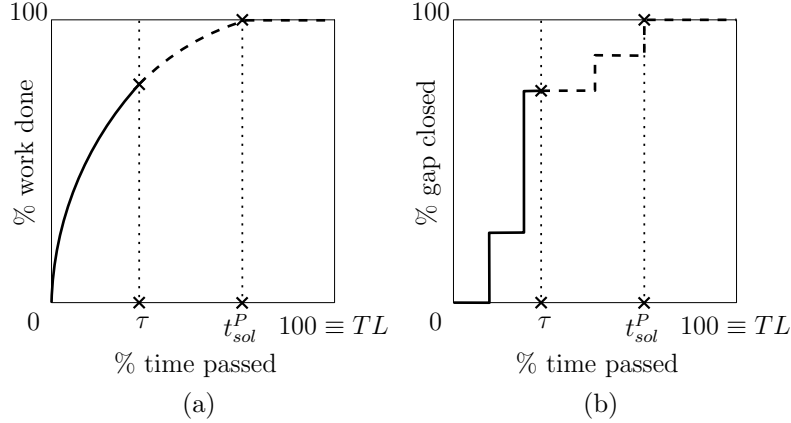
Figure 5.2 (a) Graphical example of "progress measure" for a triplet (TL, ρ, P); we assume a smooth behavior for drawing purposes. The observed portion of the resolution (up to time τ) is drawn in solid. (b) If we were to measure the progress by looking at the % of gap closed only, we would draw a step-wise linear function.

5.3.1 Sequence classification

The sequential character of B&B makes it natural to think about the partial resolution of P as a progressive stream of information and events. In the MILP context, it appears reasonable to discretize the time dimension by considering information being retrieved at every node of the B&B tree, starting from the root and up to the last one being processed before time τ (say η). In other words, one could describe the output of a MILP run with a multivariate time series $\mathcal{S}_{TL,\rho,P}$,

$$
\begin{aligned}
\mathcal{S}_{TL,\rho,P} = \Big\{ \; & (N^1, \langle v_1^1, \cdots, v_s^1 \rangle), \\
& (N^2, \langle v_1^2, \cdots, v_s^2 \rangle), \\
& \quad\vdots \\
& (N^\eta, \langle v_1^\eta, \cdots, v_s^\eta \rangle) \; \Big\},
\end{aligned}
\tag{5.3}
$$

a sequence of vectors $v^k \in \mathbb{R}^s$, each carrying information about the optimization state at node N^k, up to η.

Classifying $\mathcal{S}_{TL,\rho,P}$ depending on P's optimization outcome can be seen as a *(conventional) sequence classification* task. Sequence classification is typically employed in genomic applications, anomaly-detection and information retrieval (see, e.g., [118–120], respectively), and generally deals with learning a *sequence classifier* for data of sequential type. Few alternatives to tackle sequence classification can be found in the literature (see [116] for a brief survey).

We opt for a feature-based approach: simply put, we transform the sequence $\mathcal{S}_{TL,\rho,P}$ into a single vector of numerical features $\Phi(TL, \rho, P) \in \mathbb{R}^d$, to which we will then apply traditional classification algorithms. In our setting, a data-point for the learning algorithm consists of a tuple $\left(\Phi(TL, \rho, P), y\right)$ with $\Phi(TL, \rho, P)$ describing the time series data $\mathcal{S}_{TL,\rho,P}$, and binary label $y \in \{0, 1\}$ assigned according to $\mathbf{1}_{\{t_{sol}^P \leq TL\}}$.

As pointed out in [116], one of the major challenges when dealing with sequence classification resides in the fact that sequence data does not come with explicit features. Moreover, feature selection is usually costly, and needs to account for an interpretable prediction. Off-the-shelf feature selection methods – like k-grams or time series shapelets – do not appear suitable to capture the special temporal nature of B&B. We will present features specifically designed for the MILP resolution process after discussing the data collection methodology.

5.4 Collecting B&B data

As we said, the B&B framework produces a lot of heterogeneous information, whose combination can provide interesting insights about the optimization status of a MILP run. Extracting data from the resolution process is allowed by means of implementing custom callbacks in the solver's APIs, and comes with some computational overhead. From an application perspective, it seems reasonable that a user might be willing to spend some additional resources in the first part of the resolution process, say up to time τ, in order to get a prediction on the more lengthy horizon of TL. Nevertheless, especially in our setting, time is important: any appreciable overhead during the run could bias the yes/no response with respect to the fixed TL, so data collection has to be as cheap as possible.

In fact, the overhead we experienced comes from the computation of few indicators and the need to interface the solver through its API. For example, extracting the number of iinf variables at every branched node cannot be done with an API method directly, so that one needs to examine feasibility statuses for all variables. However, the same indicator would come almost for free if implemented internally, on the solver's side: the value of iinf at every node is available and systematically printed in the resolution log.

To comply with the need of collecting non-biased data – and certain that a data collection procedure implemented internally on the solver side would incur in much less overhead than that experienced by any user dealing with its interfaces – we devise a two-step proof-of-concept implementation. We use CPLEX 12.7.1 as solver, together with its Python API. Given (TL, ρ, P), we perform

1. *Label computation*: run P with time-limit TL, and determine a label for the run by

checking if $t_{sol}^P \leq TL$. During the run record η^P, the number of nodes processed up to time τ.

2. *Data collection*: run again P (the deterministic run of Step 1 can be reproduced by setting the same random seed), and actively collect data during the optimization, up to η^P nodes.

Having detached data collection from label computation, we do not need to worry anymore about the overhead incurred in Step 2, nor about the integrity of the labeled data; the produced sequence $\mathcal{S}_{TL,\rho,P}$ records the real "work done" up to the sought fraction ρ of TL.

5.4.1 Producing diversification

For fixed TL and ρ, a data-point corresponds to a single run of a problem P. The need of a reasonable amount of data for applying ML thus requires many MILP instances – definitely more than those currently part of MILP libraries (see Section 5.6). Instead of resorting to random problems generation, we try to create additional data from existing benchmark instances.

A first general diversification of data from the same problem P can be produced exploiting the so-called *performance variability* of MILPs [17]. Perturbations can be obtained simply by setting different random seeds in the solver, to obtain diverse runs of P. Other diversification schemes, specific to our setting, consist in varying the main parameters TL and ρ. In particular, one could (*i*) vary TL and keep ρ fixed, and/or (*ii*) vary ρ and keep TL fixed. Intuitively, approach (*i*) seems more promising at generating heterogeneous points: a change of TL allows for a sensible re-scaling of τ as well, potentially producing data labeled differently, despite coming from the same problem P. We graphically describe this intuition in Figure 5.3.

Having discussed how to produce and collect valuable MILP time series data, we now turn to the task of handling it, in order to craft a vector of features.

5.5 Feature design

We undertake a feature-based approach for sequence classification, and transform MILP sequential data $\mathcal{S}_{TL,\rho,P}$ into a single vector of features $\Phi(TL, \rho, P) \in \mathbb{R}^d$, to be fed as input to traditional classification algorithms. As already mentioned, feature selection is not a straightforward process when dealing with serial data, especially if one wants to retain a
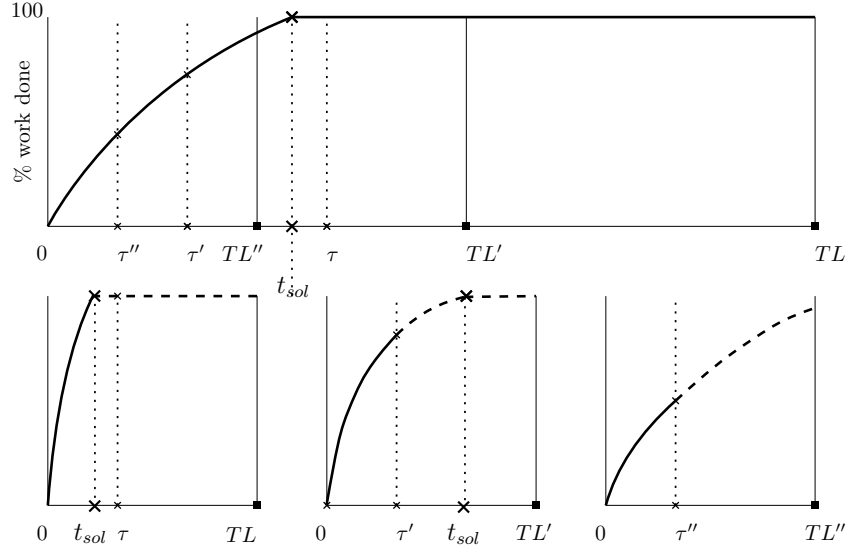
Figure 5.3 Graphical example of approach (i) to obtain multiple data-points from fixed (ρ, P), varying TL. The run of P is represented at the top. Below, using additional TL', TL'' we get $\left(\Phi(TL, \rho, P), 1\right)$, $\left(\Phi(TL', \rho', P), 1\right)$ and $\left(\Phi(TL'', \rho'', P), 0\right)$. The observed portions of the resolution process are drawn in solid.

certain degree of interpretability. We rely on MILP domain-knowledge to define features that shall encompass the optimization progress encoded in $\mathcal{S}_{TL,\rho,P}$.

In practice, we extract 25 raw numerical attributes from each callback call during Step 2 of our data collection procedure, i.e., each vector v^k of $\mathcal{S}_{TL,\rho,P}$ has dimension 25. Note, however, that the length η of the series varies considerably across instances and seeds, ranging between a few dozens and hundreds of thousands. At each branched node of the tree we collect information about the general state of the optimization (e.g., gap, value of incumbent and best bound, total number of processed nodes and count of simplex iterations performed), together with node-specific data (e.g., current node LP objective value, number of iinf in the LP solution, node depth). At few points in time, we extract information about the list of nodes left (e.g., its length, the maximum and minimum objective estimates, and the number of nodes attaining them). Data traditionally reported in the solver's log are included in these 25 attributes.

Let us point out a few remarks on the nature of the extracted B&B data, and on the guidelines that should be observed to transform them into MILP "progress measures".

1. Some pieces of raw information already describe the *global* optimization state, and can be considered in all respects as "progress measures" for the MILP resolution. An example in this sense is provided by the *gap* measure: the last datum collected about

Table 5.1 Description of the 37 features employed for learning experiments.

#	Group name	Features general description
7	Last observed global measures	Gap; ratio between best bound and incumbent; fraction of nodes left attaining max (resp. min) objective estimate; ratio between max (resp. min) estimate across nodes left and incumbent; primal-dual integral [121].
4	Nodes left and pruned, iterations count	Throughput of pruned nodes; ratio between nodes pruned and nodes left; last measure of nodes left over max observed one; throughput of simplex iterations.
4	Node LP integer infeasibilities (iinf)	Max (resp. min, avg) number of observed iinf over $|\mathcal{I}|$; fraction of nodes with iinf below 5% quantile value.
5	Incumbent	Throughput of incumbent updates (i.e., frequency); average improvement (resp. distance) of updates normalized by incumbent value (resp. total # of nodes); distance from last observed update over the average one; was an incumbent found before an integer feasible node (boolean)?
4	Best bound	Throughput of best bound updates (i.e., frequency); average improvement (resp. distance) of updates normalized by best bound value (resp. total # of nodes); distance from last observed update over the average one.
3	Node LP objective	Fraction of nodes with objective above the 95% quantile value; differences in absolute value between quantile threshold and global bounds.
4	Node LP fixed variables	Fraction of max (resp. min) observed # of fixed variables; fraction of nodes with # of fixed variables above 95% quantile value; distance from last observed peak over total # of nodes.
6	Depth and tree traversal	Ratio between max observed depth and # of processed nodes; ratio between height of last full level (resp. waist) and maximal depth [48]; maximal and average length of backtracks; frequency of backtracks in the traversal.

the gap refers to the entire resolution process up to that point, and can be used directly as feature in $\Phi(TL, \rho, P)$.

2. Some other information is instead *local*, referring to a particular node LP, and need to be embedded and interpreted within a broader and global context. For example, a single datum about the *depth* of a node is not informative of the tree evolution, but combined depth data can provide indications about the tree profile (e.g., in terms of maximal depth, width and full levels, see [48]), as well as describe dives and backtracks in the traversal.

3. Some traits are global (in the sense that they refer to the totality of the optimization process), but are not significant if taken individually. This is the case, for example, of data about the global bounds values, which present themselves as a crude sequence of decreasing (or increasing) scalar values. Measuring their development and changes, instead, can be more informative of the optimization progress.

4. Finally, the wide range of MILP benchmark instances requires features to be comparable

across the dataset. For example, exact values linked to parameters $(c, A, b, |\mathcal{I}|)$ and solutions should be avoided. Global counters, e.g., the number of processed nodes, should be used to rescale other indicators, in order not to affect the learning process (and subsequent data normalizations) with data of different magnitudes.

With these guidelines in mind, by means of combining different raw indicators with each other and interpreting them from a development perspective, we design (and select) 37 features to represent the MILP progress. We describe the features set in Table 5.1.

Besides the canonical use of statistical functions (like max, min, average) to synthesize some serial information, and the use of *throughputs* measures (e.g., to infer the rates at which nodes are processed and pruned), we apply our domain-knowledge to summarize the optimization progress. For example, we tackle measures that can vary significantly even between consecutive nodes in the B&B tree, but for which we are interested in localizing extreme behaviors only, by employing quantile values as statistically meaningful thresholds. We use them to track peaks for values of node LP *objective*, number of *iinf* and number of *fixed variables*. Instead, for data that is updating throughout the optimization process (e.g., for *incumbent* and *best bound* values), we focus on interpreting their changes in time, deduce how often and how distant are updates happening, and what is their average improvement.

5.6   Experimental results

Dataset composition and setup   We employ instances of MIPLIB 2010 [16] and [122] for our experiments. An assessment of the distribution of solving times seemed necessary in order to produce a balanced and meaningful dataset. Evaluation runs with 10 different seeds on the MIPLIB 2010 *Benchmark* set suggested the use of $TL \in \{3600, 2400, 1200\}$ seconds. A projection of the resulting labels distribution was performed, to select $\rho = 0.2$ (i.e., we stop the observation after 20% of TL).

To build our dataset, we collect B&B data from the following MILP problems:

- `Benchmark78`: 78 instances from MIPLIB 2010 *Benchmark* set (problems belonging to *Infeasible* and *Primal* subsets are removed, since they do not appear meaningful for our question);

- `Challenge160`: 160 problems from MIPLIB 2010 *Challenge* set (with *Infeasible* and *Primal* removed);

- `Mittelmann48`: 48 instances from H. Mittelmann *MILPlib* collection [122].

Problems in `Benchmark78` and `Mittelmann48` are solved with three different random seeds, while those in `Challenge160` with a single one. As expected, `Mittelmann48` runs are very short, with few cases of time-limiting problems. Counterbalancing this effect, the majority of instances in `Challenge160` cannot be solved within 1h time-limit; `Benchmark78` run times are distributed more evenly. All MILP runs were performed on a cluster of 640 48-cores machines, each equipped with a 2.1GHz Intel Platinum 8160F "Skylake" processor and 192 GB of RAM. Apart from time-limit specifications, we do not modify the solver's default setting; in particular, we leave in place CPLEX default presolve, cuts and primal heuristics.

The heterogeneity of the collected time series data makes necessary a thorough phase of data cleaning and scaling. We discard troublesome runs to get 1315 data-points, which then reduce to 970 after computing the hand-crafted features and performing basic data cleaning (data with missing values are removed). Note that a single MILP problem can generate up to 9 different data-points, given the variations in seeds and time-limits used. In the final dataset of 970 points, Class 1 (Class 0) represents the 65.6% (34.4%) of the total; a snapshot of the dataset composition is given in Table 5.2.

Train and test splits In order to account for the different composition of MILP libraries and the role of performance variability, we define and try three different ways of splitting our data into training and test set.

(a) Non-homog. split   (b) Homogeneous split   (c) Random split

Benchmark78   Challenge160   Mittelmann48
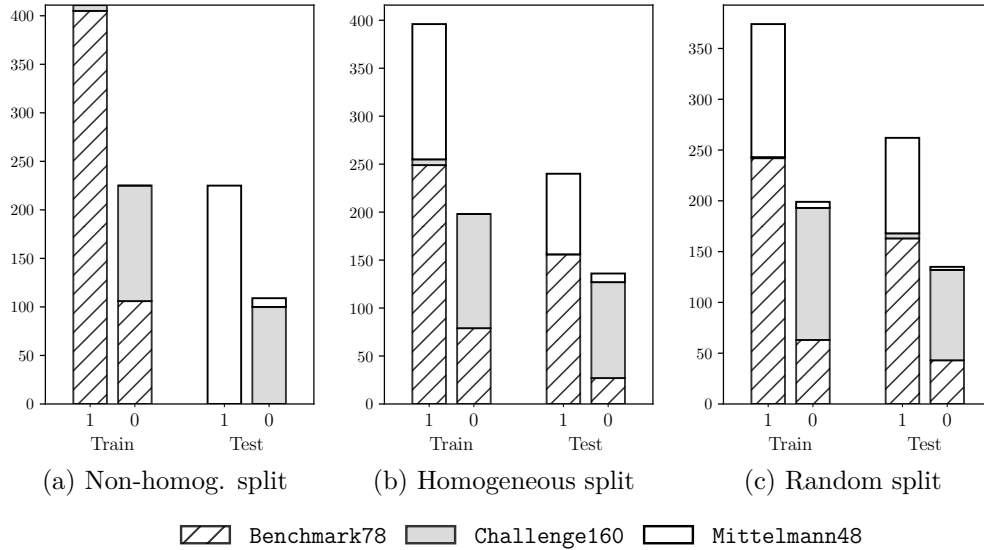
Figure 5.4 Training and test set composition with respect to different labels and MILP libraries, reported for the three considered train-test splits.

Table 5.2 Dataset composition in terms of labels and original MILP libraries.

	Class 0	Class 1	**Total (%)**
Benchmark78	106	405	511 (52.7)
Challenge160	219	6	225 (23.2)
Mittelmann48	9	225	234 (24.1)
Total (%)	334 (34.4)	636 (65.6)	**970**

1. *Non-homogeneous* split: data-points from `Benchmark78` are used for training, while those from `Mittelmann48` for test; data from `Challenge160` is divided between train and test, taking care of keeping together points arising from the same MILP instance.

2. *Homogeneous* split: both training and test sets are built using a share of each dataset. Again, points arising from the same instance are kept together.

3. *Random* split: data from all runs are mixed together and randomly split. In this case, points that originated from the same MILP instance can appear in both training and test sets.

Proportions between training and test set are roughly maintained around a 60%-40% repartition, with slight variations across splits. Figure 5.4 illustrates the datasets composition in more detail.

5.6.1 Learning experiments

We train and test five different learning models, namely, Logistic Regression (LR), Support Vector Machines (SVM) with RBF kernel [112], Random Forest (RF) [113], Extremely Randomized Trees (ExT) [88], and Multi-Layer Perceptron (MLP) [22]. All algorithms are compared against a dummy classifier (dum) following a stratified strategy, i.e., predicting by respecting the class distribution. The learning phase is implemented entirely in Python with Scikit-learn [63], and run on a PC with Intel Core i5, 2.3 GHz and 8 GB of memory. Each feature is normalized to have a mean of 0 and a standard deviation of 1, and each experiment comprises a training phase with 3-fold cross validation to grid-search hyper-parameters, and a test phase on the neutral test set.

Results Table 5.3 reports the standard performance measures for binary classification: for all classifiers we compare accuracy, precision, recall and f1-score, the last three metrics averaged between classes and weighted by supports. Overall, RF and ExT are the best

Table 5.3 Classification results for the three considered train-test split settings; we report accuracy (acc), precision (pre), recall (rec) and f1-score (f1). Measures are rounded to the second decimal; best scores and classifiers are bold-faced.

	acc	pre	rec	f1	acc	pre	rec	f1	acc	pre	rec	f1
dum	0.55	0.56	0.55	0.56	0.59	0.58	0.59	0.59	0.57	0.57	0.57	0.57
LR	0.94	0.94	0.94	0.94	0.90	0.91	0.90	0.90	0.93	0.93	0.93	0.93
SVM	0.94	0.94	0.94	0.94	0.91	0.91	0.91	0.91	0.94	0.94	0.94	0.94
RF	**0.96**	**0.96**	0.96	**0.96**	0.94	0.94	0.94	0.94	**0.94**	**0.95**	**0.94**	**0.94**
ExT	0.96	0.96	**0.96**	0.96	**0.95**	**0.95**	**0.95**	**0.95**	0.93	0.94	0.93	0.93
MLP	0.91	0.92	0.91	0.90	0.86	0.86	0.86	0.85	0.93	0.93	0.93	0.93
	(a) Non-homogeneous split				(b) Homogeneous split				(c) Random split			

Table 5.4 Confusion matrices (w/o normalization) for RF in different split settings. Note that support sizes are varying.

		Predicted					Predicted					Predicted		
		0	**1**	support			**0**	**1**	support			**0**	**1**	support
True	**0**	100	9	109	True	**0**	125	11	136	True	**0**	130	5	135
	1	3	222	225		**1**	12	228	240		**1**	18	244	262
	(a) Non-homogeneous split				(b) Homogeneous split				(c) Random split			

performing models, with SVM following close behind. We additionally report confusion matrices for RF in Table 5.4. The high accuracy scores obtained in all three train-test settings attest that there is indeed a statistical pattern to be learned during MILP resolution, and that the designed features are capturing it.

Taking a closer look at class-specific precision and recall scores, we note distinct behaviors with respect to different train-test splits. In particular, models in the *Non-homogeneous* case present a sensitivity (i.e., recall for Class 1) being higher than specificity, accompanied by high precision for Class 0. The trend is much less accentuated in the *Homogeneous* setting, and blurs completely (if not reverses itself) in the *Random* one. An explanation of these behaviors could be linked to the intrinsic difference in composition of the MILP libraries employed for the experiments. In fact, instances in `Benchmark78` do not exhibit clear-cut behaviors as those in `Challenge160` and `Mittelmann48`. Finally, the fact of *Random* being the setting in which MLP is best performing might be a sign of the model being able to recognize akin data-points arising from the same instance (now scattered in both training and test set), and thus linked to the presence of problems with low variability scores.

Table 5.5 Subset of features appearing in the top-10s for RF: scores are averaged among split cases; features marked with ∗ appear in the top-10 of each setting.

Rank	Score (avg)		Feature description
1	0.1856	∗	Throughput of pruned nodes (over total # of processed ones)
2	0.1839	∗	Ratio between pruned nodes and last measured number of nodes left
3	0.0805	∗	Last measured number of nodes left over maximal number of nodes left observed
4	0.0758	∗	Fraction of nodes left attaining max objective estimate
5	0.0632	∗	Fraction of nodes left attaining min objective estimate
6	0.0622	∗	Frequency of backtracks
7	0.0453	∗	Throughput of best bound updates
8	0.0324	∗	Last measured gap
9	0.0196		Ratio between last measured best bound and incumbent
10	0.0181		Maximal length of observed backtracks
11	0.0165		Difference in absolute value between objective 5% quantile threshold and best bound
12	0.0164		Distance from last observed best bound update over the average one

Feature analysis   Our best performing methods, RF and ExT, have the advantage of interpretability. We employ feature scores returned by Scikit-learn, measuring the mean decrease in impurity [114], to provide a first evaluation of those factors that proved valuable for the predictions. We look at the sets of top-10 scoring features for RF, for each train-test split case, and note a very stable scoring pattern: 8 features appear in the top-10 of each setting, and a total of 12 different features covers the three top rankings. We report them in Table 5.5, where scores have been averaged among cases. In particular, throughputs and trends of nodes pruned, processed and left seem to be crucial for proper classification. Information on the proportions of nodes attaining maximum and minimum objective estimates within the list of nodes left is also valuable. Indeed, such estimates at the frontier of the B&B tree are somehow quantifying the amount of work to be done to close the upper and lower bounds in the remaining subtrees, and hence measuring the "difficulty" of what is yet to be explored. Together with the gap, few top-ranked features focus on dives and backtracks happened during the traversal, while few others on best bound updates. Note that, despite having provided the same set of features to capture updates of incumbent and best bound, only those relative to the latter are top-ranked by the algorithm. This is in line with the composition of MILP benchmarking libraries and the experience of MILP practitioners, who often witness slow B&B searches due to difficulty in improving the LP (dual) bound.

5.7 Conclusions and outlook

We propose a learning approach to predict the outcome of a general MILP problem after only a share of the available computing time has passed. We summarize the sequential MILP resolution process with hand-crafted features, and successfully classify it with traditional learning models. In particular, our novel features can be applied to any type of MILP instance, and hence used in future application of ML for B&B studies, making this work of interest for a wide audience. Our positive results show that there is indeed a pattern to be learned across MILP instances, and represent (to the best of our knowledge) the first structural statistical use of the data provided by the solver throughout the resolution. The proposed framework could be readily implemented internally on the solver side, in order to strategically specialize the optimization process on the fly, before timing out, providing better options for the user. In other words, an early detection of a potential time out can trigger algorithmic changes that, in turn, could prevent such a time out to happen. The developed setting can be extended in a number of different directions. We plan to deepen data analysis – possibly augmenting our dataset – and frame the role of performance variability in the learning process. It would be interesting to consider other ways to tackle sequence classification, e.g., by following a pattern-based approach.

CHAPTER 6   PARAMETERIZING BRANCH-AND-BOUND SEARCH TREES TO LEARN BRANCHING POLICIES

Authors: Giulia Zarpellon, Jason Jo, Andrea Lodi and Yoshua Bengio[1]
Submitted for conference publication.[2]

Abstract   Branch and Bound (B&B) is the exact tree search method typically used to solve Mixed-Integer Linear Programming problems (MILPs). Learning branching policies for MILP has become an active research area, with most works proposing to imitate the strong branching rule and specialize it to distinct classes of problems. We aim instead at learning a policy that generalizes across heterogeneous MILPs: our main hypothesis is that parameterizing the state of the B&B search tree can significantly aid this type of generalization. We propose a novel imitation learning framework, and introduce new input features and architectures to represent branching. Experiments on MILP benchmark instances clearly show the advantages of incorporating to a baseline model an explicit parameterization of the state of the search tree to modulate the branching decisions. The resulting policy reaches higher accuracy than the baseline, and on average explores smaller B&B trees, while effectively allowing generalization to generic unseen instances.

6.1   Introduction

Many problems arising from transportation, healthcare, energy and logistics can be formulated as Mixed-Integer Linear Programming (MILP) problems, i.e., optimization problems in which some decision variables represent discrete or indivisible choices. A MILP is written as

$$\min_{x} \{c^{T}x : Ax \geq b, x \geq 0, \ x_{i} \in \mathbb{Z} \ \forall i \in \mathcal{I}\}, \tag{6.1}$$

where $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^{m}$, $c, x \in \mathbb{R}^{n}$ and $\mathcal{I} \subseteq \{1, \ldots, n\}$ is the set of indices of variables that are required to be integral, while the other ones can be real-valued. Note that one can consider a MILP as defined by $(c, A, b, \mathcal{I})$; we do not assume any special combinatorial structure on the parameters c, A, b. While MILPs are in general $\mathcal{NP}$-hard, MILP solvers underwent dramatic improvements over the last decades [1,2] and now achieve high-performance on a wide range of problems. The fundamental component of any modern MILP solver is Branch and Bound (B&B) [3], an *exact* tree search method. Following a divide-and-conquer approach, B&B

---

[1]Authors are listed by relative contribution, as is standard practice in Computer Science journals and conferences.
[2]Available at [30].

partitions the search space by branching on variables' values and smartly uses bounds from problem relaxations to prune unpromising regions from the tree. The B&B algorithm actually relies on expertly-crafted *heuristic* rules for its two most fundamental decisions: *branching variable selection* (BVS) and *node selection.* In particular, BVS has been shown to be a crucial factor for B&B's success [1], and will be the main focus of the present article.

Understanding why B&B works has been called "one of the mysteries of computational complexity theory" [123], and there currently is no mathematical theory of branching; to the best of our knowledge, the only attempt in formalizing BVS is the recent work of [74]. One central reason why B&B is difficult to formalize resides in its inherent exponential nature: millions of BVS decisions could be needed to solve a MILP, and a single bad one could result in a doubled tree size and no improvement in the search. Such a complex and data-rich setting, paired with a lack of formal understanding, makes B&B an appealing ground for machine learning (ML) techniques, which have lately been thriving in discrete optimization [6]. In particular, there has been substantial effort towards "learning to branch", i.e., in using ML methods to learn BVS policies [25].

Up to now, most works in this area of research focused on learning branching policies by supervision or imitation of *strong branching* (SB), a valid but expensive heuristic scheme (see Sections 6.2 and 6.5 for more details). While [38] propose to explicitly learn SB scores by regression, [37] formulate BVS as a ranking problem and learn instance-specific proxies of SB. In a different vein, [39] suggest to leverage existing scoring rules by learning weights to combine them, and perform experiments on special classes of synthetic problems. The latest contribution we know of to "learning to branch" [40] frames BVS as a classification problem on SB expert decisions, and employs a graph-convolutional neural network to represent MILPs. The resulting branching policies improve on the solver by successfully specializing SB to different classes of generated combinatorial optimization problems. Specifically, the attained generalization ability is to similar MILP instances (within the same class), possibly larger in formulation size.

The present work seeks a different (somehow complementary) type of generalization for a branching policy, namely across *heterogeneous* MILPs, i.e., across problems not belonging to the same combinatorial class, without any restriction on the formulation's structure and size. To achieve this goal, we parameterize BVS in terms of B&B search trees. On the one hand, information about the state of the B&B tree – abundant yet mostly unexploited by MILP solvers – was already shown to be useful to learn resolution patterns shared across general MILPs [29]. On the other hand, the state of the search tree ought to have a central role in BVS – which ultimately decides how the tree is expanded and hence how the search

itself proceeds. In practice, B&B continually interacts with other algorithmic components of the solver to effectively search the decision tree, and some algorithmic decisions may be triggered depending on which phase the optimization is in [14]. In a highly integrated framework, a branching variable should thus be selected among the candidates based on its role in the search and its various components. Indeed, state-of-the-art heuristic branching schemes employ properties of the tree to make BVS decisions, and the B&B method equipped with such branching rules has proven to be successful across widely heterogeneous instances.

Motivated by these considerations, our main hypothesis is that MILPs share a higher order structure in the space of B&B search trees, and parameterized BVS policies should learn in this representational space. We setup a novel learning framework to investigate this idea. First of all, there is no natural input representation of this underlying space. Our first contribution is to craft input features of the variables that are candidates for branching: we aim at representing their broad roles in the search and their dynamic evolution. The dimensionality of such descriptions naturally changes with the number of candidate variables at every BVS step. The deep neural network (DNN) architecture that we propose learns a baseline branching policy (NoTree) from the candidate variables' representations and effectively deals with varying input dimensions. Taking this idea further, we suggest that an explicit representation of the *state of the search tree* should condition the branching criteria, in order for it to flexibly adapt to the tree evolution. We contribute such tree-state parameterization, and incorporate it to the baseline architecture to provide context over the candidate variables at each given branching step. In the resulting policy (TreeGate) the tree state acts as a control mechanism to drive a top-down modulation (specifically, feature gating) of the highly mutable space of candidate variables representations. In this sense, we learn branching from parameterizations of B&B search trees that are shared among general MILPs. To the best of our knowledge, the present work is the first attempt in the "learning to branch" literature to represent B&B search trees for branching, and to establish such a broad generalization paradigm covering many classes of MILPs. We envision a future combination of our framework on generic MILPs with more structure-based ones, such as that of [40], to leverage the strengths of both approaches.

We perform imitation learning (IL) experiments on a curated dataset of heterogeneous instances from standard MILP benchmarks: the selected problems belong to various special classes, are different in structure and size, and give rise to diverse search trees. We employ as expert rule the default branching scheme of the optimization solver SCIP [124], to which our framework is integrated. Machine learning experimental results clearly show the advantage of the policy employing the tree state (TreeGate) over the baseline one (NoTree), the former achieving a 19% improvement in test accuracy. The evaluation of the trained policies

in the solver also supports our idea that representing B&B search trees enables learning to branch across generic MILP instances: the best TreeGate policy explores on average trees with 14.9% less nodes than the best NoTree one; measured over test instances only, this gap increases to 27%. In addition, when plugged in the solver both learned policies compare well with state-of-the-art branching rules.

6.2 Background

Simply put, the B&B algorithm iteratively partitions the solution space of a MILP (6.1) into sub-problems, which are mapped to nodes of a binary decision tree. At each node, integrality requirements for variables in $\mathcal{I}$ are dropped, and a linear programming (LP) (continuous) relaxation of the problem is solved to provide a valid lower bound to the optimal value of (6.1). When the solution x^* of a node LP relaxation violates the integrality of some variables in $\mathcal{I}$, that node is further partitioned into two children by *branching on a fractional variable*. Formally, $\mathcal{C} = \{i \in \mathcal{I} : x_i^* \notin \mathbb{Z}\}$ defines the index set of *candidate variables* for branching at that node. The BVS problem consists in selecting a variable $j \in \mathcal{C}$ in order to *branch* on it, i.e., create child nodes according to the split

$$x_j \leq \lfloor x_j^* \rfloor \ \lor \ x_j \geq \lceil x_j^* \rceil. \tag{6.2}$$

Child nodes inherit a lower bound estimate from their parent, while (6.2) ensures x^* is removed from their solution spaces. After extending the tree, the algorithm moves on to select a new *open node*, i.e., a leaf yet to be explored (node selection): a new relaxation is solved, and new branchings happen. When x^* satisfies integrality requirements, then it is actually feasible for (6.1), and its value provides a valid upper bound to the optimal one. Maintaining global upper and lower bounds allows one to prune large portions of the search space. During the search, *final leaf nodes* are created in three possible ways: by integrality, when the relaxed solution is feasible for (6.1); by infeasibility of the sub-problem; by bounds, when the comparison of the node's lower bound to the global upper one proves that its sub-tree is not worth exploring. An optimality certificate is reached when the global bounds converge. See [2,7] for details on B&B and its combination with other components of a MILP solver.

Branching rules Usually, candidates are evaluated with respect to some scoring function, and j is chosen for branching as the (or a) score-maximizing variable. The most used criterion in BVS measures variables depending on the improvement of the lower bound in their (prospective) child nodes. The *strong branching* (SB) rule [71] explicitly computes bound

gains for $\mathcal{C}$. The procedure is expensive, but experimentally realizes trees with the least number of nodes. Instead, *pseudo-cost* (PC) [72] maintains a history of variables' branchings, averaging past improvements to get a proxy for the expected gain. Fast in evaluation, PC can behave badly due to uninitialization, so combinations of SB with PC have been developed. In *reliability branching*, SB is performed until PC scores for a variable are deemed reliable proxies of bound improvements. In *hybrid branching* [73], PC scores are combined via a weighted sum with other criteria borrowed from the CSP and SAT communities (on inference and conflict clauses). Extensive literature has been produced on BVS schemes [70], and many other scoring criteria have been proposed; some of them are surveyed by [25] from a machine learning perspective.

State-of-the-art branching rules can in fact be interpreted as mechanisms to score variables based on their effectiveness in different search components. While hybrid branching explicitly combines five scores reflecting variables' behaviors in different search tasks, the evaluation performed by SB and PC can also be seen as a measure of how effective a variable is – in the single task of improving the bound from one parent node to its children. Besides, one can assume that the importance of different search functionalities should change dynamically during the tree exploration.[3] In this sense, our approach aims at learning a branching rule that takes into account variables' roles in the search and the tree evolution itself to perform a more flexible BVS, adapted to the search stages.

6.3   Parameterizing B&B search trees

The central idea of our framework is to learn BVS by means of parameterizing the underlying space of B&B search trees. We believe this space can represent the complexity and the dynamism of branching in a way that is shared across heterogeneous problems. However, there are no natural parameterizations of BVS or B&B search trees. To this end, our contribution is two-fold: 1) we propose hand-crafted input features to describe candidate variables in terms of their roles in the B&B process, and explicitly encode a "tree state" to provide a richer context to variable selection; 2) we design novel DNN architectures to integrate these inputs and learn BVS policies.

6.3.1   Hand-crafted input features

At each branching step t, we represent the set of variables that are candidates for branching by an input matrix $C_t \in \mathbb{R}^{25 \times |\mathcal{C}_t|}$. To capture the multiple roles of a variable throughout the

---

[3]Indeed, a "dynamic factor" takes care of adjusting hybrid weights in the default branching scheme of SCIP (see `https://scip.zib.de/doc-6.0.0/html/branch__relpscost_8c_source.php#l00524`).
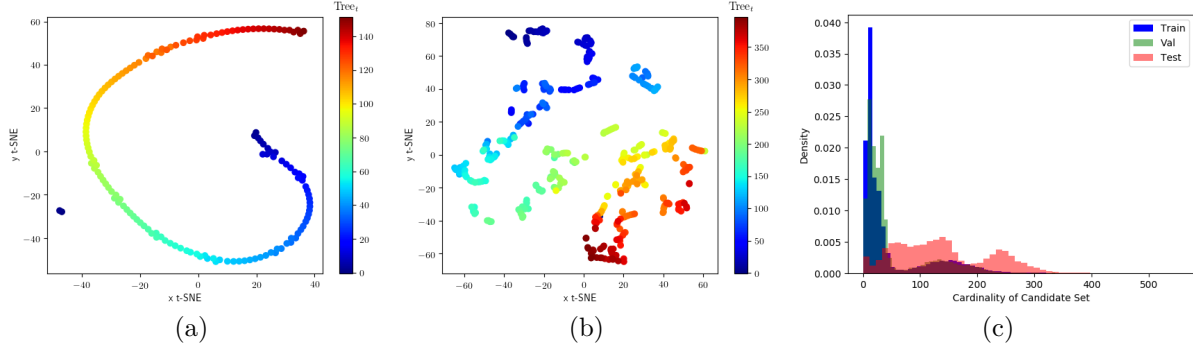
Figure 6.1 The evolution of the $Tree_t$ representation throughout the B&B search is synthesized by t-SNE plots (perplexity=5), for instances (a) eil33-2 and (b) seymour1. (c) Histogram of $|\mathcal{C}_t|$ in train, validation and test data.

search, we describe each candidate $x_j, j \in \mathcal{C}_t$ in terms of its bounds and solution value in the current sub-problem. We also feature statistics of a variable's participation in various search components (e.g., inference, conflicts, implications) and in past branchings. In particular, the scores that are used in the SCIP default hybrid-branching formula are part of C_t.

Additionally, we create a separate parameterization $Tree_t \in \mathbb{R}^{61}$ to describe the state of the search tree. We record information of the current local node in terms of its depth and the quality of its bound. We also consider the growth rate and composition of the tree (explored, open, final leaf nodes), the evolution of global bounds, statistics on feasible solutions and multiple other scores, aggregated over variables. Statistics on bound estimates and depths of the open nodes complete the parameterization of $Tree_t$.

All features are designed to capture the dynamics of the B&B process linked to BVS decisions, and are efficiently gathered through a customized version of PySCIPOpt [125]. Note that $\{C_t, Tree_t\}$ are defined in a way that is not explicitly dependent on the parameters of each instance $(c, A, b, \mathcal{I})$. Even though C_t naturally changes its dimensionality at each BVS step t depending on the highly variable $\mathcal{C}_t$, the fixed lengths of the vectors enable training among branching sets of different sizes (see 6.3.2). The representations evolve with the search: t-SNE plots [126] in Figures 6.1a and 6.1b synthesize the evolution of the tree state representation $Tree_t$ throughout the B&B search, for two different MILP instances. The pictures clearly show the high heterogeneity of the branching data across different search stages. A detailed description of the hand-crafted input features is reported in Appendix C.
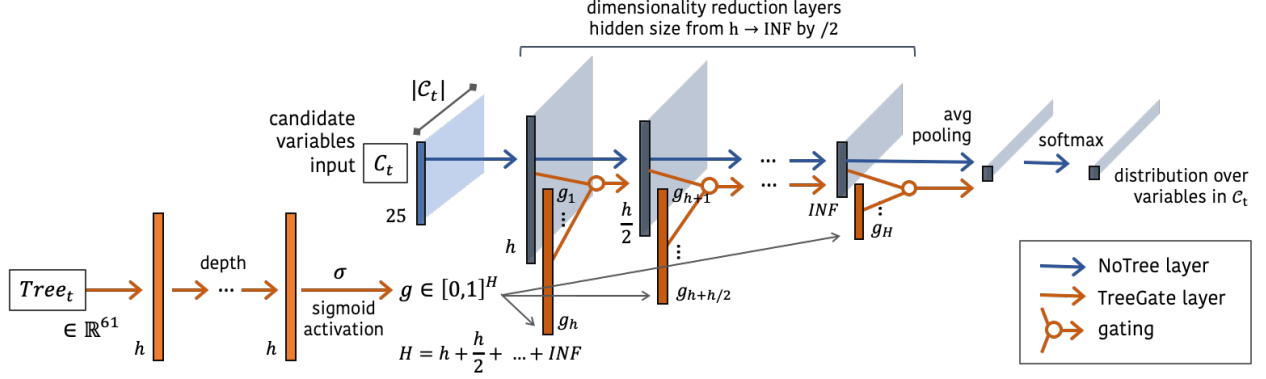
Figure 6.2 Candidate variables input C_t is processed by NoTree layers (in blue) to select a variable for branching. For the TreeGate model, the $Tree_t$ input is first embedded and then utilized in gating layers (in orange) on the candidates' representations.

6.3.2 Architectures to model branching

We use parameterizations C_t as inputs for a baseline DNN architecture (NoTree). Referring to Figure 6.2, the 25-feature input of a candidate variable is first embedded into a representation with hidden size h; subsequently, multiple layers reduce the dimensionality from h to an infimum INF by halving it at each step. The vector of length INF is then compressed by global average pooling into a single scalar. The $|C_t|$ dimension of C_t is conceived (and implemented) as a "batch dimension": this makes it possible to handle branching sets of varying sizes, still allowing the parameters of the nets to be shared across problems. Ultimately, a *softmax* layer yields a probability distribution over the candidate set C_t, according to which a variable is selected for branching.

We incorporate the tree-state input to the baseline architecture in order to provide a search-based context over the mutable branching sets. Practically, $Tree_t$ is embedded in a series of subsequent layers with hidden size h. The output of a final sigmoid activation is $g \in [0,1]^H$, where $H = h + h/2 + \cdots + INF$ denotes the total number of units of the NoTree layers. Separate chunks of g are used to modulate by feature gating the representations of NoTree: $[g_1, \ldots, g_h]$ controls features at the first embedding, $[g_{h+1}, \ldots, g_{h+h/2}]$ acts at the second layer, $\ldots$, and so on, until exhausting $[g_{H-INF}, \ldots, g_H]$ with the last layer prior the average pooling. In other words, g is used as a control mechanism on variables parameterization, gating their features via a learned tree-based signal. The resulting network (TreeGate) models the high-level idea that a branching scheme should adapt to the tree evolution, with variables' selection criteria dynamically changing throughout the tree search.

6.4 Experiments

MILP dataset and solver setting Despite MILP libraries containing hundreds of instances, not all of them appear viable for our setting, and a careful dataset curation is needed. On the one hand, comparing the behavior of different branching policies becomes easier (and results are clearer) when the explored trees are manageable in size and the problems can be consistently solved to optimality. On the other hand, standard MILP collections comprise very challenging instances, and are compiled to be the ongoing benchmark for advances in MILP research. In our ML context, it does not seem necessary to introduce extra challenges on the MILP side. We hence curate a heterogeneous collection of 27 problems from different real-world MILP benchmark libraries [16, 23, 81, 122], focusing on instances whose tree exploration is on average relatively contained (in the tens/hundreds of thousands nodes, maximum) and whose optimal value is known. We partition our selection into 19 train and 8 test problems. A complete list of instances is reported in Table 6.1, while we refer to Appendix C for other details.

We use SCIP 6.0.1 [124]. Modifying the solver configuration is common practice in BVS literature [15], especially in a proof-of-concept setting in which our work is positioned. To reduce the effects of the other solver's components on BVS, we work with a configuration specifically designed to fairly compare the performance of different branching rules [127]. In particular, we allow presolve and cut separation routines, and disable all primal heuristics. For each problem we provide the known optimal solution value as cutoff, and disable few other parameters associated with SB side-effects. We also enforce a limit of one hour on the resolution time. The same setting is used for both data collection and policies' evaluation. Further details on the solver parameters and hardware settings are reported in Appendix C.

Data collection and split We collect IL training data from SCIP roll-outs, gathering inputs $\mathrm{x}_t = \{C_t, \mathit{Tree}_t\}$ and corresponding branching decisions (labels) $\mathrm{y}_t \in \mathcal{C}_t$. Our expert branching scheme is the default one of SCIP, `relpscost`, i.e., a reliability version of hybrid branching. Given that each branching decision gives rise to a single data-point $(\mathrm{x}_t, \mathrm{y}_t)$, and that the search trees of the selected MILP instances are not extremely big, one needs to augment the data. We proceed in two ways.

(i) First, we naturally exploit the so-called *performance variability* of MILPs [17]. To obtain perturbed tree searches of the same instance, we set five different random seeds in the solver, $s \in \{0, \dots, 4\}$ to control variables' permutations.

(ii) Second, we diversify B&B explorations by letting a `random` branching scheme run for

Table 6.1 List of MILP instances used in train and test sets.

TRAIN	air04, air05, dcmulti, eil33-2, istanbul-no-cutoff, l152lav, lseu, misc03, neos20, neos21, neos-476283, neos648910, pp08aCUTS, rmatr100-p10, rmatr100-p5, sp150x300d, stein27, swath1, vpm2
TEST	map18, mine-166-5, neos11, neos18, ns1830653, nu25-pr12, rail507, seymour1

the first k nodes, before switching to SCIP default branching rule and starting data collection. The motivation behind this type of augmentation is to gather input states that are unlikely to be observed by an expert rule [42]: inputs collected from a default SCIP run will differ from those confronted by a trained IL policy at test time, given that a single divergent BVS may cascade and result in qualitatively very different B&B trees. We use $k \in \{0, 1, 5, 10, 15\}$, where the case of $k = 0$ corresponds to a run in which no random branching is performed (i.e., `relpscost` is used from the beginning of the search). We apply this second type of augmentation to train instances only.

One can quantify MILP variability by computing the coefficient of variation of the performance measurements [16]; we report such variability scores *VS* for all our instances in Appendix C, using the total number of nodes as performance measure, across the five runs of SCIP performed on different seeds, as in (i). The observed coefficients range in $[0.03, 1.70]$: the majority of the instances presents a variability of at least 0.20, confirming (i) as an effective way of diversifying our dataset. The effect of k initial random branchings is also analyzed and reported in Appendix C. Generally, the size of the explored trees grows with k, i.e., initial random branchings affect the nodes' count for worse – though the opposite can also happen in few cases. The coefficients of variation of the nodes shifted geometric means across different k's range in $[0.07, 0.79]$ in the training set, so (ii) also appears effective for data augmentation.

The final composition of train, validation and test splits is summarized in Table 6.2. In particular, train and validation data come from the same subset of 19 instances, with validation being performed on branchings from a different random seed. Instead, the test set contains data-points from 8 separate MILPs, using augmentation of type (i) only.

An important measure to analyze the dataset is given by the size of the candidate sets (i.e., the varying dimensionality of the C_t inputs) contained in each split. Figure 6.1c shows histograms for $|\mathcal{C}_t|$ in each subset. While in train and validation the candidate set sizes are mostly concentrated in the $[0, 50]$ range, the test set has a very different distribution of $|\mathcal{C}_t|$, and in particular one with a longer tail (over 300). In this sense, the test instances

Table 6.2 For train, validation and test set splits we report the total number of data-points as well as the seed-k pairs (s, k) from which they are obtained.

	Total	(s, k) **pairs**
Train	85,533	$\{0, 1, 2, 3\} \times \{0, 1, 5, 10, 15\}$
Validation	14,413	$\{4\} \times \{0, 1, 5, 10, 15\}$
Test	28,307	$\{0, 1, 2, 3, 4\} \times \{0\}$

Table 6.3 Best trained NoTree and TreeGate models. For each policy, we report the corresponding hyper-parameters, top-1 and top-5 test and validation accuracy scores, as well as shifted geometric means of B&B nodes for the entire MILP dataset (ALL), TRAIN and TEST instances. The top-5 accuracy (acc@5) measures whether the expert label was among the top-5 choices as identified by a policy's probability distribution over the candidate variables. Policies selected as best ones are boldfaced.

Policy	h / d / LR	Test acc@1 (@5)	Val acc@1 (@5)	ALL	TRAIN	TEST
NoTree	32 / − / 0.0001	68.37 (91.43)	75.40 (95.23)	1341.72	859.17	3695.04
	64 / − / 0.0001	67.05 (89.18)	76.45 (95.11)	1363.73	847.63	4010.65
	128 / − / 0.0001	65.44 (90.21)	76.77 (95.66)	1454.20	875.19	4601.72
	128 / − / 0.001	**64.02 (88.51)**	**77.69 (95.88)**	**1241.79**	**834.40**	**3068.96**
	256 / − / 0.0001	64.59 (90.13)	77.29 (96.08)	1279.18	731.16	4491.64
TreeGate	**64 / 5 / 0.01**	**83.70 (95.83)**	**84.33 (96.60)**	**1056.79**	**759.94**	**2239.47**
	256 / 2 / 0.001	83.69 (95.18)	84.10 (96.42)	1135.28	822.80	2369.35
	32 / 3 / 0.01	83.31 (95.72)	84.02 (96.50)	1188.48	809.18	2849.28
	128 / 5 / 0.001	81.61 (95.81)	84.96 (96.74)	1127.31	771.60	2666.73

present never-seen branching data gathered from heterogeneous MILPs, and we test the generalization of our policies to entirely unknown and larger branching sets.

IL optimization   We train both IL policies using ADAM [128] with default $\beta_1 = 0.9, \beta_2 = 0.999$, and weight decay 1e−5. Our hyper-parameter search spans: learning rate $LR \in \{0.01, 0.001, 0.0001\}$, hidden size $h \in \{32, 64, 128, 256\}$, and depth $d \in \{2, 3, 5\}$. The factor by which units of NoTree are reduced is 2, and we fix $INF = 8$. We use PyTorch [129] to train the models for 40 epochs, reducing LR by a factor of 10 at epochs 20 and 30.

6.4.1   Results

In our context, standard IL metrics are informative yet incomplete measures of performance for evaluating a learned BVS model, and one also cares about assessing the policies' behaviors when plugged in the solver environment. This is why in order to determine the best NoTree and TreeGate policies we take into account both types of evaluations. We first select few

policies based on their test accuracy score; next, we specify them as custom branching rules in SCIP and perform full roll-outs on the entire MILP dataset, over five random seeds (i.e., 135 evaluations each). To summarize the policies' performance in the solver, we compute the shifted geometric mean (with a shift of 100) of the total number of explored nodes, over the 135 B&B executions (ALL), and restricted to TRAIN and TEST instances.

Both types of metrics are reported in Table 6.3, together with the policies' hyper-parameters. Incorporating an explicit parameterization of the state of the search tree to modulate BVS clearly aids generalization: the advantage of TreeGate over NoTree is evident in all metrics, and across multiple trained policies. In particular, the top-1 test accuracy averages at 65.90 ± 1.6 for the NoTree models, while TreeGate ones score at 83.08 ± 0.86; the gap in validation accuracy is also significant. In terms of B&B roll-outs, NoTree models explore on average 1336.12 ± 73.32 nodes, against the 1126.97 ± 46.85 of TreeGate ones. What we observe is that best test accuracy does not necessarily translate into best solver performance. The NoTree policy with the best solver performance exhibits an approximately 4% gap from the optimal top-1 test accuracy model, but an improvement over 7% in solver performance. We select as best policies those yielding the best nodes average over the entire dataset. In the case of TreeGate, the best model corresponds to that realizing the best top-1 test accuracy (83.70%), and brings a 19% (resp. 7%) improvement over the NoTree policy, in top-1 (resp. top-5) test accuracy. Learning curves and further details can be found in Appendix C.

In solver evaluations, NoTree and TreeGate are also compared to SCIP default branching scheme `relpscost`, PC branching `pscost` and a `random` one. Additionally, we compute the *fair* number of nodes [127]. This measure accounts for those nodes that are processed as side-effects of SB-like explorations, specifically looking at domain reduction and cutoffs counts. In other words, the fair number of nodes distinguishes tree-size reductions due to better branching from those obtained by SB side-effects. Note that for rules that do not involve any SB, the fair number of nodes and the usual nodes' count coincide, so we only report it for the `relpscost` policy. The selected solver parametric setting (the same as the one used for data collection) allows a meaningful computation of the fair number of nodes, and a honest comparison of branching schemes.

Both NoTree and TreeGate policies are able to solve all instances within the 1h time-limit, like `relpscost`. In contrast, `random` hits the limit on 4 instances (17 times in total) while `pscost` does so on one instance only (neos18), a single time. Table 6.4 reports the nodes' means for every MILP instance (over five runs), as well as measures aggregated over train and test sets, and the entire dataset. In aggregation, TreeGate is always better than NoTree, the former exploring on average trees with 14.9% less nodes. This gap becomes more pronounced when

Table 6.4 Total number of nodes explored by learned and SCIP policies, in shifted geometric means over 5 runs on seeds $\{0, \ldots, 4\}$. We mark with * the cases in which time-limits were hit. For `relpscost`, we also compute the *fair* number of nodes. The percentage difference between NoTree and TreeGate is reported. Aggregated measures are over the entire dataset (ALL), as well as over TRAIN and TEST sets.

Instance	Set	NoTree	TreeGate	% diff	random	pscost	relpscost (fair)
	ALL	1241.79	1056.79	**-14.90**	6580.79	1471.61	286.15 (719.20)
	TRAIN	834.40	759.94	**-8.92**	2516.04	884.37	182.27 (558.34)
	TEST	3068.96	2239.47	**-27.03**	61828.29	4674.34	712.77 (1276.76)
air04	train	645.99	536.07	-17.02	6677.96	777.65	8.19 (114.39)
air05	train	789.70	516.06	-34.65	12685.83	1158.89	60.25 (277.22)
dcmulti	train	203.53	187.49	-7.88	599.12	122.39	9.38 (68.30)
eil33-2	train	7780.85	8767.27	12.68	12502.02	8337.63	583.34 (9668.71)
istanbul-no-cutoff	train	447.26	543.71	21.56	1085.16	613.68	242.39 (328.25)
l152lav	train	621.82	687.91	10.63	6800.06	964.53	10.14 (250.04)
lseu	train	372.67	396.71	6.45	396.73	375.31	148.99 (389.88)
misc03	train	241.40	158.39	-34.39	118.37	151.07	12.11 (294.11)
neos20	train	2062.23	1962.95	-4.81	10049.15	2730.01	200.26 (612.75)
neos21	train	1401.84	1319.73	-5.86	7016.55	1501.54	668.44 (1455.29)
neos648910	train	140.05	175.82	25.54	1763.05	1519.01	39.83 (166.53)
neos-476283	train	13759.59	6356.81	-53.80	*94411.77	2072.84	204.88 (744.65)
pp08aCUTS	train	267.86	293.74	9.66	337.76	271.92	69.66 (350.21)
rmatr100-p5	train	443.35	460.48	3.86	1802.38	451.71	411.93 (785.15)
rmatr100-p10	train	908.27	906.04	-0.25	4950.77	894.65	806.35 (1214.76)
sp150x300d	train	868.60	785.27	-9.59	1413.64	991.52	182.22 (300.42)
stein27	train	1371.44	1146.79	-16.38	1378.91	1322.36	926.82 (1111.25)
swath1	train	1173.14	1165.39	-0.66	1429.21	1107.52	298.58 (2485.63)
vpm2	train	589.03	440.74	-25.18	594.62	546.45	199.46 (463.12)
map18	test	457.89	575.92	25.78	11655.33	1025.74	270.25 (441.18)
mine-166-5	test	3438.44	4996.48	45.31	*389437.62	4190.41	175.10 (600.22)
neos11	test	3326.32	3223.46	-3.09	29949.69	4728.49	2618.27 (5468.05)
neos18	test	15611.63	10373.80	-33.55	228715.62	*133437.40	2439.29 (5774.36)
ns1830653	test	6422.37	5812.03	-9.50	288489.30	12307.90	3489.07 (4311.84)
nu25-pr12	test	357.00	86.80	-75.69	1658.41	342.47	21.39 (105.61)
rail507	test	9623.05	3779.05	-60.73	*80575.84	4259.98	543.39 (859.37)
seymour1	test	3202.20	1646.82	-48.57	*167725.65	3521.47	866.32 (1096.67)

measured over test instances only (27%), indicating the advantage of TreeGate over NoTree when exploring unseen data. Results are less clear-cut from an instance-wise perspective, with neither policy emerging as an absolute winner. Nonetheless, TreeGate is at least 10% (resp. 25%) better than NoTree on 10 (resp. 8) instances, while the opposite only happens 6 (resp. 3) times. In this sense, the reductions in tree sizes achieved by TreeGate are overall more pronounced.

In addition, learned policies compare well to other branching rules: both NoTree and Tree-Gate are substantially better than `random` across all instances, and always better than `pscost`

in aggregated measures. Only on one instance both policies are much worse than `pscost` (neos-476283). As expected, `relpscost` still realizes the smallest trees, but comparisons in terms of fair number of nodes are nonetheless positive: on 11 instances, at least one among NoTree and TreeGate explores less nodes than the `relpscost` fair number. In general, the policies realize tree sizes that are comparable to the SCIP default ones, when SB side effects are taken into account.

6.5 Related work

Among the first attempts in "learning to branch", [38] perform regression to learn proxies of SB scores. Instead, [37] propose to learn the ranking associated with such scores, and train instance-specific models (that are not end-to-end policies) via SVM^{rank}. Also [130] treat BVS as a ranking problem, and specialize their models to the combinatorial class of time-dependent traveling salesman problems. More recently, the work of [39] learns mixtures of existing branching schemes for different classes of synthetic problems, focusing on sample complexity guarantees. Similarly to us, the latest contribution to "learning to branch" [40] frames BVS as classification of expert branching decisions and employs IL to learn a branching policy. However, their expert of choice is SB, and MILPs are represented via a graph-convolutional neural network (GCNN) that models the variable-constraint structure expressed by the parameter matrix A. The resulting policies are specializations of SB that appear to effectively capture structural characteristics of some classes of combinatorial optimization problems, and are able to generalize to larger formulations from the same distribution (i.e., within the same combinatorial class). It is not obvious that the GCNN could effectively generalize across heterogeneous problems, given the policy in [40] was only tested to solve bigger instances for which small analogs were available during training.

Still concerning the B&B framework, [42] employ IL to learn a heuristic and class-specific node selection policy, and categorize B&B input features. An RL approach for node selection can be found in [41], where a Multi-Armed Bandit is used to model the tree search; some complexity and scaling issues of B&B are also presented.

Feature gating has a long and successful history in machine learning, ranging from LSTMs [131] to GRUs [132], and we refer to [133] for a survey. The idea of using a tree state to drive a feature gating of the branching variables is an example of top-down modulation, which has been shown to perform well in other deep learning applications [134–136]. With respect to learning across non-static action spaces, the most similar to our work is [137], which is in the continual learning setting. Unlike the traditional Markov Decision Process formulation of reinforcement learning (RL), the input to our policies is not a generic state but rather

includes a parameterized hand-crafted input representation of the available actions, thus continual learning is not a relevant concern for our framework. Other related works from the RL setting learn action space representations [138, 139], but they both assume that the action space is *static* across RL episodes, while in contrast the action space of BVS changes dynamically with $|\mathcal{C}_t|$.

6.6 Conclusions and future directions

Branching variable selection is a crucial factor in B&B success, and we setup a novel imitation learning framework to address it. In particular, we seek to learn branching policies that generalize across heterogeneous MILPs, regardless of the instances' structure and formulation size. In doing so, we undertake a step towards a broader type of generalization. The novelty of our approach is relevant for both the ML and the MILP worlds. On the one hand, we develop parameterizations of the candidate variables and of the search trees, and design a DNN architecture that handles candidate sets of varying size. On the other hand, the data encoded in our $Tree_t$ parameterization is not currently exploited by state-of-the-art MILP solvers, but we show that this type of information could indeed help in adapting the branching criteria to different search dynamics. Our results on MILP benchmark instances clearly demonstrate the advantage of incorporating a search-tree context to modulate BVS and aid generalization to heterogeneous problems, in terms of both better test accuracy and smaller explored B&B trees.

There surely are additional improvements to be gained by continuing to explore IL methods for branching. [40] have shown a correlation between the structure of MILPs (captured by the GCNN) and at least one of the main ingredients of the expert we use, namely the bound improvement due to BVS. In this work, we exploit instead representations of general B&B trees, and both priors may be required to fully match the expert performance.

However, quantifying the goodness of branching policies and B&B search trees remains hard due to the complexity and exponentiality of the B&B system. In the IL setting this translates into not being able to assess the impact of a misclassified BVS in the subsequent tree exploration. In fact, the MILP domain expertise suggests that at any given branching step there is no such thing as a single best branching decision, but rather groups of variables on which one should branch [78]. In other words, there is no branching ground truth, and the quality of branching certainly resides in effective BVS sequences. For these reasons, barring a mathematical breakthrough for the theory of branching, we believe there can be much more innovation in future explorations of RL approaches for BVS. Within the RL paradigm the focus would shift to learning branching sequences and partial trees explorations, by means

of heterogeneous reward signals that could allow to better approach the diverse performance goals one practically aims at when solving MILPs. These are important factors in "learning to branch" which cannot be expressed in IL terms. Indeed, the idea and the benefits of using an explicit parameterization of the state of B&B search trees – which we demonstrated in the IL setting – could be expanded even more in the RL one, for both state representations and the design of branching rewards.

Acknowledgements

CHAPTER 7   GENERAL DISCUSSION

The four works presented through Chapters 3 to 6 contribute in varied ways to the general objective of this dissertation: they tackle different ideas on the theme of applying ML algorithms to MIP algorithmic design, and each of them develops an experimental framework that is specific to its research question. Although one may see these works as self-contained and isolated contributions, they in fact naturally influence and echo each other, both methodologically and in terms of content.

The in-depth analysis of B&C and its functioning in the MIP solver, which has been the foundation of our survey (Chapter 3, [25]), provided useful tools to explore the research questions of both Chapters 5 and 6. In particular, the study of state-of-the-art branching heuristics and first ML-based approaches matured into our own contribution to the *learning to branch* theme. The literature covered in Chapter 3 also gave us a solid starting point to develop input features for the tasks of predicting solver's time-limits and learning to branch. Indeed, both sets of attributes try to capture similar things: in Chapter 5 the stream of sequential data from the first part of the optimization is condensed into a single vector of hand-crafted features to represent the progress in the search; in Chapter 6, instead, at each branching step the data is not only gathered, but also manipulated and fed to DNN models, which then automatically synthetize it to decide on variable selection.

The nice-to-have properties of branching rules identified in Section 3.4 significantly influenced our design of a branching policy in Chapter 6, notably towards the idea that a branching scheme should be adaptive not only with respect to different instances, but throughout the evolution of the tree search as well. Besides, the positive results and the analysis of feature importance in Chapter 5 had a substantial role in motivating the main hypothesis of Chapter 6 – namely, that representing the state of the B&B search tree could aid generalization over heterogeneous MILP instances. The concept of a shared pattern in MILP resolution that could be leveraged to condition the search naturally translated into the TreeGate architecture of Chapter 6, which employs a *tree state* to modulate the choice of branching variables and yields improved generalization performance.

From the methodological standpoint, each work builds on the others' "lessons learned" to solidify the process of integration of ML into the MIP solver. While our early works [26] and [29] focus on simple proof-of-concept settings and experiment with off-the-shelf supervised learning models, the methodologies established in Chapters 4 and 6 are more tailored to their research questions, and ultimately lead to a tighter combination of predictions and

optimization. We briefly highlight some transversal themes that emerged and matured during this incremental process, which we believe could serve as keys to read our works as a whole, and to develop new ones.

The importance of introducing domain knowledge and optimization priors in the learning pipeline played out across all our contributions, and feature design proved to be a critical step in this sense. As opposed to other typical ML applications (e.g., image recognition), there is no natural representation of MIP instances, and one needs to carefully represent data for the task at hand. To provide comprehensive representations of our research questions via feature engineering, we tried to think about which factors played a role in the decisions we wanted to predict, and to implement hand-crafted feature vectors that incorporated not only mathematical but also algorithmic clues. The trade-off between expressiveness and conciseness of feature representations was also repeatedly addressed. In Chapter 5, for example, the overhead of computing some attributes potentially interferes with the learning framework. When predicting whether to linearize MIQPs in Chapter 4, instead, we resorted to approximating an important feature from the root node relaxations, and similarly cut off other traits that did not appear viable for online computation. Finally, for the problem of learning to branch (Chapter 6) signals from the solver environment need to be retrieved efficiently, given that they are extracted at every branching node.

Another explored trade-off was the one between accuracy and explainability of the learned predictors, i.e., the ability to evaluate the quality of predictions from the perspective of optimization performance. We soon realized how this aspect is of paramount importance when applying ML to practical MIP settings: predictions catalyze tangible actions in the solver, whose impact on the optimization needs to be quantified, trusted and (as far as possible) understood. Practically, though, these concepts are challenging to work out, and we hope that future research will try to approach them in a principled way (see also Section 8.2). In our works, the difference between *learning metrics* and *performance metrics* particularly stands out in Chapter 4, where additional scores are developed to provide information on the impact of misclassification in terms of runtime, i.e., the performance metric that is used to define targets. The use of weights in the training phase is also a way of introducing some prior about the sought optimization performance of the predictors. In the imitation learning setup of Chapter 6, this concern is reflected in the fact that it is hard to relate the performance of a learned branching policy with that of the expert (in our case, SCIP default behavior). Indeed, best test accuracy does not necessarily translate into smallest B&B trees, and the relationship between solver performance and the nature of local minima in the ML optimization landscape ought to be further explored.

Given the role of data in capturing the essence of a ML research question, a considerable effort in all our works was spent in identifying, generating or selecting MIP instances for learning experiments. In applications where there is a one-to-one correspondence between a data-point and a MIP problem, benchmark libraries could result insufficient: this was the case for MIQPs in Chapter 4, where we opted for a dataset of both real-world and synthetic instances. In Chapters 5 and 6, we could employ diversification schemes to obtain multiple samples from a single MILP. Performance variability [17] appeared particularly helpful in this task, though the effect of such randomizations on data and the learning process is not easily framed. For some other learning tasks (e.g., learning to branch) a MILP problem may naturally provide many data-points. Despite data being abundant, however, other considerations may become necessary in order to curate a dataset that is usable and meaningful for the given experimental setting, as was the case in Chapter 6. Note also that in contexts like branching undersampling procedures might not be easily defined, given that samples from the B&B search are not independent and identically distributed in the classical sense.

Finally, decisions on dataset composition ultimately depend on one's generalization target. Clearly, generalizing to a wide range of instances seems more challenging, as ML models may fail when evaluated on never-seen instances that are too different from those used for training. For ML to succeed, one should leverage the *commonalities* of problems within a selected distribution, e.g., by effectively representing them in the input features. This was the effort undertaken in both Chapters 4 and 6; experiments of Chapter 5, on the other hand, have been essential in uncovering some algorithmic traits that might be shared in the optimization of heterogeneous MILP instances. Additionally, one may need to consider different levels of generalization. This is true for sequential applications like branching, where a policy also needs to generalize to internal states of the B&B search, which, as we saw in Chapter 6, can look quite dissimilar. While all our contributions target very broad generalization scopes and work with instances from general-purpose MIP benchmark libraries, a definitive meaning of generalization for MIP applications remains hard to pinpoint.

CHAPTER 8   CONCLUSIONS AND RECOMMENDATIONS

This thesis discussed and proposed applications of statistical learning methods to the MIP framework, specifically in the context of MIP algorithmic design. After a brief summary of the presented contributions, we conclude this dissertation by highlighting their limitations and proposing some directions for related future research.

8.1   Summary of works

Motivated by the need of organizing early works on the theme of ML in B&B, we presented a survey on learning and branching (Chapter 3). After including an overview of ML techniques and transversal methodological concepts, the work revises the B&B paradigm and state-of-the-art approaches for variable and node selection. The analysis of both topics sets out with the discussion of previous MIP literature contributions, which we interpret as forerunners of ML-based works. Identifying their assumptions and concerns provides an original canvas to discuss the more recent learned methods that tackle B&B.

Our second contribution deals with the design of an algorithmic decision for MIQPs (Chapter 4). In particular, we addressed with ML the question of whether to linearize the quadratic part of convex MIQPs when solving them with CPLEX. The developed classification framework aims at tightly integrating the optimization knowledge in the learning pipeline, e.g., by means of using a context-specific scoring function and sample weights, as well as alternative evaluation metrics that reflect the prediction quality in terms of solver performance. As experiments practically led to the deployment of a classifier to decide on MIQP linearization (CPLEX v12.10.0), the work also contributes a reference methodology for the combination of ML and MIP technologies.

In Chapter 5 we discussed a learning approach to predict the optimization outcome of general MILPs after only a fraction of the available computing time has passed. We looked at the evolution of partial B&C trees and applied a feature-based sequence classification approach: first the sequential optimization process is summarized via hand-crafted features, and traditional learning models are then used to cast binary predictions. Despite being limited to an offline exploratory setting, the work opens new opportunities for MILP algorithmic design. In particular, the achieved results support the idea that MILPs share algorithmic behaviors in general-purpose B&C solving, and that such patterns can be described and identified with a statistical use of the data from optimization.

These intuitions motivated the main hypothesis of our fourth contribution (Chapter 6). With the aim of learning a branching policy that generalizes across heterogeneous MILP instances, we developed a novel IL framework in which both features and DNN architectures are designed to represent the evolving state of the B&C search, and its potential influence on variable selection. Experiments on MILP benchmark instances demonstrate that incorporating such broader, tree-related context leads to branching policies with better test accuracy that also on average explore smaller trees. Indeed, the resulting policies appear to effectively adapt not only to unseen problems – thus establishing the sought type of generalization range – but to the evolution of the B&C tree as well.

8.2 Limitations and future research

The works presented in this thesis belong to the body of initial explorations on the theme of combining ML and MIP, and more generally of augmenting discrete optimization algorithms with data-driven, AI-based approaches. Our contributions add to the growing literature of this recent and already fruitful research area; by proposing new insights, methodologies and points of view, they also expose original venues for future research.

While the field just moved its first steps, some themes already appear decisive for advancing towards a tighter integration of ML and MIP technologies. In particular, common practices for development, data and benchmarking will likely be key aspects to ensure a steady progress in the area. After all, both MIP and ML development histories demonstrate how much benchmark libraries and tools can be instrumental to significant advances – consider the role played by, e.g., MIPLIB [16], ImageNet [140] and OpenAI Gym [141] in establishing common, recognized baselines. The current development of Ecole [142], a library that aims to expose (and experiment with) control problems in MIP solvers, represent an encouraging step in this direction. Together with the adoption of shared ML practices, the curation of fixed MIP datasets for ML experiments would greatly help future reproducibility, comparison and improvement of learned predictors, e.g., of the classifier developed in Chapter 4.

Despite our efforts to interpret features and condition the *optimization performance* of predictions (as discussed in Chapter 7), a general limitation of our contributions is the use of ML models as mainly black-box tools. In this sense, the explainable combination of ML in MIP seems a far-reaching goal: being able to interpret and trust predictors could allow them to be implemented in solvers more swiftly, thus bridging the gap between making predictions and prescribing decisions in MIP algorithms. For example, to develop the framework of Chapter 5 to its full potential, predictions on MILP outcome should be cast on-the-fly, and follow up with prescriptions of concrete algorithmic actions to be taken by the solver

in order to flexibly guide the optimization. The use of interpretable rule-based classification systems, for instance, could allow predictions to also carry in themselves practical indications for the solver, not only tailoring the resolution process but enhancing its comprehension as well. A deep reflection on MILP solving becomes essential to build such schemes, as one would need to identify simple, frequent, and distinctive behaviors in B&C and their required solver adjustments, and encode them in a learning environment.

We already discussed in Chapter 6 how RL appears to be the fit paradigm for *learning to branch*. With this end in mind, future work could be focused on the design of reward mechanisms for variable selection. On the one hand, one practically aims at diverse performance goals when solving MILPs (e.g., early feasibility, substantial pruning, fast proof of optimality), and our idea of parameterizing the state of B&B search trees seems a promising starting point to develop reward signals that better approach them. On the other hand, reward design is in itself a complex and active field of RL research, and all the issues identified in [21] very well apply to the B&B context. Indeed, experimental settings like the ones B&B provides are full of challenges for both the ML and the MIP communities, and exploring this synergy could bring truly interdisciplinary research outcomes. New computational studies on B&C explorations would be needed to support the task of reward design; statistically analyzing how search trees (and their properties) evolve could provide new empirical insights, and also help with the identification of trends in MILP optimization behavior, as mentioned above. At the same time, the idea of parameterizing the non-static action space of branching candidates which we devised for the DNN architectures of Chapter 6 could potentially be re-purposed and studied for other ML applications.

Finally, the works of this thesis have been developed with heterogeneous MIPs and general-purpose MIP solvers in mind. We hope that our investigation allowed to appreciate the interesting questions and unique challenges posed by this setting, as we believe there is room for improving MIP algorithmic design via statistical learning methods.

REFERENCES

[1] T. Achterberg and R. Wunderling, *Mixed Integer Programming: Analyzing 12 Years of Progress.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 449–481.

[2] A. Lodi, "Mixed integer programming computation," in *50 Years of Integer Programming 1958-2008*, M. Jünger, T. Liebling, D. Naddef, G. Nemhauser, W. Pulleyblank, G. Reinelt, G. Rinaldi, and L. Wolsey, Eds. Springer Berlin Heidelberg, 2009, pp. 619–645.

[3] A. Land and A. Doig, "An automatic method of solving discrete programming problems," *Econometrica*, vol. 28, pp. 497–520, 1960.

[4] A. Lodi, "The heuristic (dark) side of MIP solvers," in *Hybrid Metaheuristics*, ser. Studies in Computational Intelligence, E.-G. Talbi, Ed. Springer Berlin Heidelberg, 2013, no. 434, pp. 273–284.

[5] T. Achterberg, "Constraint integer programming," Ph.D. dissertation, Technical University of Berlin, 2007.

[6] Y. Bengio, A. Lodi, and A. Prouvost, "Machine learning for combinatorial optimization: a methodological tour d'horizon," *arXiv preprint 1811.06128*, 2018.

[7] L. A. Wolsey, *Integer programming.* New York, NY, USA: Wiley-Interscience, 1998.

[8] M. Conforti, G. Cornuéjols, and G. Zambelli, *Integer Programming*, ser. Graduate Texts in Mathematics. Springer International Publishing, 2014.

[9] R. Gomory, "An algorithm for the mixed integer problem," The Rand Corporation, Tech. Rep. RM-2597, 1960.

[10] M. Jörg, "k-disjunctive cuts and a finite cutting plane algorithm for general mixed integer linear programs," *arXiv preprint 0707.3945*, 2007.

[11] S. Dash, N. B. Dobbs, O. Günlük, T. J. Nowicki, and G. M. Świrszcz, "Lattice-free sets, multi-branch split disjunctions, and mixed-integer programming," *Mathematical Programming*, vol. 145, no. 1-2, pp. 483–508, 2014.

[12] M. Padberg and G. Rinaldi, "A branch and cut algorithm for the resolution of large-scale symmetric traveling salesmen problems," *SIAM Review*, vol. 33, no. 1, pp. 60–100, 1991.

[13] M. Fischetti and A. Lodi, "Heuristics in mixed integer programming," *Wiley Encyclopedia of Operations Research and Management Science*, 2010.

[14] T. Berthold, G. Hendel, and T. Koch, "From feasibility to improvement to proof: three phases of solving mixed-integer programs," *Optimization Methods and Software*, vol. 33, no. 3, pp. 499 – 517, 2017.

[15] J. T. Linderoth and M. W. P. Savelsbergh, "A computational study of search strategies for mixed integer programming," *INFORMS Journal on Computing*, vol. 11, no. 2, pp. 173–187, 1999.

[16] T. Koch, T. Achterberg, E. Andersen, O. Bastert, T. Berthold, R. E. Bixby, E. Danna, G. Gamrath, A. M. Gleixner, S. Heinz, A. Lodi, H. Mittelmann, T. Ralphs, D. Salvagnin, D. E. Steffy, and K. Wolter, "MIPLIB 2010," *Mathematical Programming Computation*, vol. 3, no. 2, pp. 103–163, 2011.

[17] A. Lodi and A. Tramontani, *Performance Variability in Mixed-Integer Programming.* INFORMS, 2013, ch. 1, pp. 1–12.

[18] M. Fischetti and M. Monaci, "Exploiting erraticism in search," *Operations Research*, vol. 62, no. 1, pp. 114–122, 2014.

[19] M. Fischetti, A. Lodi, M. Monaci, D. Salvagnin, and A. Tramontani, "Improving branch-and-cut performance by random sampling," *Mathematical Programming Computation*, vol. 8, no. 1, pp. 113–132, 2016.

[20] C. M. Bishop, *Pattern Recognition and Machine Learning*, ser. Information Science and Statistics. Springer-Verlag New York, Inc., 2006.

[21] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction.* MIT press, 1998.

[22] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning.* MIT Press, 2016. [Online]. Available: http://www.deeplearningbook.org

[23] A. Gleixner, G. Hendel, G. Gamrath, T. Achterberg, M. Bastubbe, T. Berthold, P. M. Christophel, K. Jarck, T. Koch, J. Linderoth, M. Lübbecke, H. D. Mittelmann, D. Ozyurt, T. K. Ralphs, D. Salvagnin, and Y. Shinano, *MIPLIB 2017: Data-Driven Compilation of the 6th Mixed-Integer Programming Library*, 2019. [Online]. Available: http://www.optimization-online.org/DB_HTML/2019/07/7285.html

[24] F. Furini, E. Traversi, P. Belotti, A. Frangioni, A. Gleixner, N. Gould, L. Liberti, A. Lodi, R. Misener, H. Mittelmann, N. Sahinidis, S. Vigerske, and A. Wiegele, "QPLIB: A library of quadratic programming instances," *Mathematical Programming Computation*, 2018.

[25] A. Lodi and G. Zarpellon, "On learning and branching: a survey," *TOP*, vol. 25, no. 2, pp. 207–236, Jul 2017.

[26] P. Bonami, A. Lodi, and G. Zarpellon, "Learning a classification of mixed-integer quadratic programming problems," in *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, W.-J. van Hoeve, Ed. Springer International Publishing, 2018, pp. 595–604.

[27] P. Bonami, A. Lodi, and G. Zarpellon, "A classifier to decide on the linearization of mixed-integer quadratic problems in CPLEX," *Optimization Online preprint*, 2020. [Online]. Available: http://www.optimization-online.org/DB_HTML/2020/03/7662.html

[28] IBM ILOG CPLEX. (2020) CPLEX Optimizer. [Online]. Available: https://www.ibm.com/analytics/cplex-optimizer

[29] M. Fischetti, A. Lodi, and G. Zarpellon, "Learning MILP resolution outcomes before reaching time-limit," in *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, L.-M. Rousseau and K. Stergiou, Eds. Cham: Springer International Publishing, 2019, pp. 275–291.

[30] G. Zarpellon, J. Jo, A. Lodi, and Y. Bengio, "Parameterizing branch-and-bound search trees to learn branching policies," *arXiv preprint 2002.05120*, 2020.

[31] Zuse Institute Berlin. (2020) The SCIP Optimization Suite. [Online]. Available: https://scip.zib.de

[32] H. H. Hoos, "Automated Algorithm Configuration and Parameter Tuning," in *Autonomous Search*, Y. Hamadi, E. Monfroy, and F. Saubion, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 37–71.

[33] B. Bischl, P. Kerschke, L. Kotthoff, M. Lindauer, Y. Malitsky, A. Fréchette, H. Hoos, F. Hutter, K. Leyton-Brown, K. Tierney, and J. Vanschoren, "ASlib: A benchmark library for algorithm selection," *Artificial Intelligence*, vol. 237, pp. 41–58, Aug. 2016.

[34] F. Hutter, H. Hoos, and K. Leyton-Brown, "Automated configuration of mixed integer programming solvers," *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pp. 186–202, 2010.

[35] F. Hutter, L. Xu, H. H. Hoos, and K. Leyton-Brown, "Algorithm runtime prediction: Methods & evaluation," *Artificial Intelligence*, vol. 206, pp. 79–111, 2014.

[36] G. Di Liberto, S. Kadioglu, K. Leo, and Y. Malitsky, "DASH: Dynamic approach for switching heuristics," *European Journal of Operational Research*, vol. 248, no. 3, pp. 943–953, 2016.

[37] E. Khalil, P. Le Bodic, L. Song, G. Nemhauser, and B. Dilkina, "Learning to branch in mixed integer programming," in *Proceedings of the 30th AAAI Conference on Artificial Intelligence*, 2016.

[38] A. Marcos Alvarez, Q. Louveaux, and L. Wehenkel, "A machine learning-based approximation of strong branching," *INFORMS Journal on Computing*, vol. 29, no. 1, pp. 185–195, 2017.

[39] M.-F. Balcan, T. Dick, T. Sandholm, and E. Vitercik, "Learning to branch," in *International Conference on Machine Learning*, 2018, pp. 344–353.

[40] M. Gasse, D. Chetelat, N. Ferroni, L. Charlin, and A. Lodi, "Exact combinatorial optimization with graph convolutional neural networks," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 15 554–15 566.

[41] A. Sabharwal, H. Samulowitz, and C. Reddy, "Guiding combinatorial optimization with uct," in *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, N. Beldiceanu, N. Jussien, and É. Pinson, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 356–361.

[42] H. He, H. Daume III, and J. M. Eisner, "Learning to search in branch and bound algorithms," in *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2014, pp. 3293–3301.

[43] E. B. Khalil, B. Dilkina, G. Nemhauser, S. Ahmed, and Y. Shao, "Learning to run heuristics in tree search," in *26th International Joint Conference on Artificial Intelligence (IJCAI)*, 2017.

[44] M. Kruber, M. E. Lübbecke, and A. Parmentier, "Learning when to use a decomposition," in *Integration of AI and OR Techniques in Constraint Programming*, D. Salvagnin and M. Lombardi, Eds.   Springer International Publishing, 2017, pp. 202–210.

[45] R. Baltean-Lugojan, R. Misener, P. Bonami, and A. Tramontani, "Strong sparse cut selection via trained neural nets for quadratic semidefinite outer-approximations," Imperial College, London, Tech. Rep., 2018.

[46] Y. Tang, S. Agrawal, and Y. Faenza, "Reinforcement learning for integer programming: Learning to cut," *arXiv preprint 1906.04859*, 2019.

[47] D. E. Knuth, "Estimating the efficiency of backtrack programs," *Mathematics of Computation*, vol. 29, no. 129, pp. 122–136, 1975.

[48] G. Cornuéjols, M. Karamanov, and Y. Li, "Early estimates of the size of branch-and-bound trees," *INFORMS Journal on Computing*, vol. 18, no. 1, pp. 86–96, 2006.

[49] A. Marcos Alvarez, L. Wehenkel, and Q. Louveaux, "Machine learning to balance the load in parallel branch-and-bound," Université de Liège, Tech. Rep., 2015. [Online]. Available: http://hdl.handle.net/2268/181086

[50] D. Anderson, G. Hendel, P. Le Bodic, and M. Viernickel, "Clairvoyant restarts in branch-and-bound search using online tree-size estimation," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, 2019, pp. 1427–1434.

[51] O. Vinyals, M. Fortunato, and N. Jaitly, "Pointer Networks," in *Advances in Neural Information Processing Systems 28*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, Eds.   Curran Associates, Inc., 2015, pp. 2692–2700.

[52] I. Bello, H. Pham, Q. V. Le, M. Norouzi, and S. Bengio, "Neural Combinatorial Optimization with Reinforcement Learning," in *International Conference on Learning Representations*, 2017. [Online]. Available: https://openreview.net/forum?id=Bk9mxlSFx

[53] H. Dai, E. B. Khalil, Y. Zhang, B. Dilkina, and L. Song, "Learning combinatorial optimization algorithms over graphs," in *Advances in Neural Information Processing Systems (NIPS)*, 2017.

[54] E. Larsen, S. Lachapelle, Y. Bengio, E. Frejinger, S. Lacoste-Julien, and A. Lodi, "Predicting solution summaries to integer linear programs under imperfect information with machine learning," *arXiv preprint 1807.11876*, 2018.

[55] M. Fischetti and M. Fraccaro, "Using OR + AI to predict the optimal production of offshore wind parks: A preliminary study," in *Optimization and Decision Science: Methodologies and Applications*, A. Sforza and C. Sterle, Eds.   Springer International Publishing, 2017, pp. 203–211.

[56] D. Bertsimas and J. Dunn, "Optimal classification trees," *Machine Learning*, vol. 106, no. 7, pp. 1039–1082, 2017.

[57] O. Günlük, J. Kalagnanam, M. Menickelly, and K. Scheinberg, "Optimal decision trees for categorical data via integer programming," *arXiv preprint 1612.03225*, 2018.

[58] S. Dash, O. Gunluk, and D. Wei, "Boolean decision rules via column generation," in *Advances in Neural Information Processing Systems*, 2018, pp. 4655–4665.

[59] B. Amos and J. Z. Kolter, "OptNet: Differentiable optimization as a layer in neural networks," in *Proceedings of the 34th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, vol. 70.   PMLR, 2017, pp. 136–145.

[60] A. Ferber, B. Wilder, B. Dilkina, and M. Tambe, "MIPaaL: Mixed integer program as a layer," in *Proceedings of the 34th AAAI Conference on Artificial Intelligence*, 2020.

[61] J. T. Linderoth and A. Lodi, "MILP software," in *Wiley Encyclopedia of Operations Research and Management Science*, J. Cochran, Ed.   John Wiley & Sons, Inc., 2011, vol. 5, pp. 3239–3248.

[62] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference and Prediction*, 2nd ed., ser. Springer series in statistics.   Springer-Verlag New York, Inc., 2009.

[63] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[64] B. Bischl, M. Lang, L. Kotthoff, J. Schiffner, J. Richter, E. Studerus, G. Casalicchio, and Z. M. Jones, "mlr: Machine learning in R," *Journal of Machine Learning Research*, vol. 17, no. 170, pp. 1–5, 2016.

[65] JuliaComputing. (2017) Machine learning and artificial intelligence. [Online]. Available: https://juliacomputing.com/domains/machine-learning.html

[66] P. Domingos, "A few useful things to know about machine learning," *Communications of the ACM*, vol. 55, no. 10, pp. 78–87, 2012.

[67] R. Bellman, *Adaptive Control Processes.* Princeton University Press, 1961.

[68] D. P. Bertsekas and J. N. Tsitsiklis, *Neuro-Dynamic Programming*, 1st ed., ser. Anthropological Field Studies. Athena Scientific, 1996.

[69] C. Szepesvári, *Algorithms for Reinforcement Learning.* Morgan & Claypool Publishers, 2010, vol. 4, no. 1.

[70] T. Achterberg, T. Koch, and A. Martin, "Branching rules revisited," *Operations Research Letters*, vol. 33, no. 1, pp. 42–54, 2005.

[71] D. Applegate, R. Bixby, V. Chvátal, and W. Cook, *The Traveling Salesman Problem. A Computational Study.* Princeton University Press, 2007.

[72] M. Benichou, J. Gauthier, P. Girodet, and G. Hentges, "Experiments in mixed-integer programming," *Mathematical Programming*, vol. 1, pp. 76–94, 1971.

[73] T. Achterberg and T. Berthold, "Hybrid Branching," in *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems.* Springer, Berlin, Heidelberg, 2009, pp. 309–311.

[74] P. Le Bodic and G. Nemhauser, "An abstract model for branching and its application to mixed integer programming," *Mathematical Programming*, pp. 1–37, 2017.

[75] A. Marcos Alvarez, "Computational and theoretical synergies between linear optimization and supervised machine learning," Ph.D. dissertation, Université de Liège, Liège, Belgique, 2016.

[76] W. Glankwamdee and J. Linderoth, "Lookahead branching for mixed integer programming," in *Twelfth INFORMS Computing Society Meeting.* INFORMS, 2011, pp. 130–150.

[77] F. K. Karzan, G. L. Nemhauser, and M. W. P. Savelsbergh, "Information-based branching schemes for binary linear mixed integer problems," *Mathematical Programming Computation*, vol. 1, no. 4, pp. 249–293, 2009.

[78] M. Fischetti and M. Monaci, "Backdoor branching," *INFORMS Journal on Computing*, vol. 25, no. 4, pp. 693–700, 2012.

[79] A. Gilpin and T. Sandholm, "Information-theoretic approaches to branching in search," *Discrete Optimization*, vol. 8, no. 2, pp. 147–159, 2011.

[80] C. E. Shannon, "A mathematical theory of communication," *The Bell System Technical Journal*, vol. 27, pp. 379–423 and 623–656, 1948.

[81] R. E. Bixby, S. Ceria, C. M. McZeal, and M. W. P. Savelsbergh, "An updated mixed integer programming library: MIPLIB 3.0," *Optima*, vol. 58, pp. 12–15, 1998.

[82] S. Kadioglu, Y. Malitsky, and M. Sellmann, "Non-model-based search guidance for set partitioning problems," in *Proceedings of the 26th AAAI Conference on Artificial Intelligence*, 2012.

[83] G. Hamerly and C. Elkan, "Learning the k in k-means," in *Neural Information Processing Systems (NIPS)*, vol. 3.   MIT Press, 2003, pp. 281–288.

[84] J. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, vol. 1, no. 14.   University of California Press, 1967, pp. 281–297.

[85] C. Ansótegui, M. Sellmann, and K. Tierney, "A gender-based genetic algorithm for the automatic configuration of algorithms," in *Principles and Practice of Constraint Programming - CP 2009: 15th International Conference.*   Springer, Berlin, Heidelberg, 2009, pp. 142–157.

[86] H. Abdi and L. J. Williams, "Principal component analysis," *Wiley Interdisciplinary Reviews: Computational Statistics*, vol. 2, no. 4, pp. 433–459, 2010.

[87] M. Fischetti and M. Monaci, "Branching on nonchimerical fractionalities," *Operations Research Letters*, vol. 40, no. 3, pp. 159–164, 2012.

[88] P. Geurts, D. Ernst, and L. Wehenkel, "Extremely randomized trees," *Machine Learning*, vol. 63, no. 1, pp. 3–42, 2006.

[89] T. Achterberg, T. Koch, and A. Martin, "MIPLIB 2003," *Operations Research Letters*, vol. 34, no. 4, pp. 361–372, 2006.

[90] A. Marcos Alvarez, L. Wehenkel, and Q. Louveaux, "Online learning for strong branching approximation in branch-and-bound," Université de Liège, Tech. Rep., 2016. [Online]. Available: http://hdl.handle.net/2268/192361

[91] J. Nocedal and S. Wright, *Numerical Optimization*, 2nd ed. New York: Springer, 2006.

[92] A. Marcos Alvarez, Q. Louveaux, and L. Wehenkel, "A supervised machine learning approach to variable branching in branch-and-bound," Université de Liège, Tech. Rep., 2014. [Online]. Available: http://hdl.handle.net/2268/167559

[93] T. Joachims, "Training linear SVMs in linear time," in *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2006, pp. 217–226.

[94] E. Khalil, "Machine learning for integer programming," in *Proceedings of the Doctoral Consortium at the Twenty-Fifth International Joint Conference on Artificial Intelligence (IJCAI)*, 2016.

[95] H. Robbins, "Some aspects of the sequential design of experiments," *Bulletin of the American Mathematical Society*, vol. 58, no. 5, pp. 527–535, 1952.

[96] COR@L, *Computational Optimization Research at Lehigh*, 2017. [Online]. Available: https://coral.ise.lehigh.edu

[97] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," in *Machine Learning: ECML 2006: 17th European Conference on Machine Learning*. Springer, Berlin, Heidelberg, 2006, pp. 282–293.

[98] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine Learning*, vol. 47, no. 2, pp. 235–256, 2002.

[99] C. Sammut, *Behavioral Cloning*. Boston, MA: Springer US, 2010, pp. 93–97.

[100] U. Syed and R. E. Schapire, "A reduction from apprenticeship learning to classification," in *Advances in Neural Information Processing Systems 23*, J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, Eds. Curran Associates, Inc., 2010, pp. 2253–2261.

[101] Gurobi. (2020) Gurobi Optimizer. [Online]. Available: http://www.gurobi.com

[102] C. Bliek, P. Bonami, and A. Lodi, "Solving mixed-integer quadratic programming problems with IBM-CPLEX: a progress report," in *Proceedings of the Twenty-Sixth RAMP Symposium*, 2014, pp. 171–180.

[103] R. Fourer, "Quadratic optimization mysteries, part 1: Two versions," dis-ORiented, March 2015. [Online]. Available: http://bob4er.blogspot.ca/2015/03/quadratic-optimization-mysteries-part-1.html

[104] J. Czyzyk, M. P. Mesnier, and J. J. Moré, "The NEOS server," *IEEE Journal on Computational Science and Engineering*, vol. 5, no. 3, pp. 68—75, 1998.

[105] E. D. Dolan, "The NEOS server 4.0 administrative guide," Mathematics and Computer Science Division, Argonne National Laboratory, Technical Memorandum ANL/MCS-TM-250, 2001.

[106] W. Gropp and J. J. Moré, "Optimization environments and the NEOS server," in *Approximation Theory and Optimization*, M. D. Buhman and A. Iserles, Eds. Cambridge University Press, 1997, pp. 167–182.

[107] G. P. McCormick, "Computability of global solutions to factorable nonconvex programs: Part I — convex underestimating problems," *Mathematical Programming*, vol. 10, no. 1, pp. 147–175, 1976.

[108] W. P. Adams, R. J. Forrester, and F. W. Glover, "Comparisons and enhancement strategies for linearizing mixed 0-1 quadratic programs," *Discrete Optimization*, vol. 1, no. 2, pp. 99 – 120, 2004.

[109] A. Billionnet, S. Elloumi, and A. Lambert, "Extending the QCR method to general mixed-integer programs," *Mathematical programming*, vol. 131, no. 1-2, pp. 381–401, 2012.

[110] T. S. Motzkin and E. G. Straus, "Maxima for graphs and a new proof of a theorem of Turán," *Canadian Journal of Mathematics*, vol. 17, pp. 533–540, 1965.

[111] MINOA, *Open-source benchmark library*, 2019 (accessed December, 2019). [Online]. Available: https://minoa-itn.fau.de/benchmark-instances/

[112] C. Cortes and V. Vapnik, "Support-vector networks," *Machine Learning*, vol. 20, no. 3, pp. 273–297, 1995.

[113] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[114] G. Louppe, "Understanding random forests: From theory to practice," Ph.D. dissertation, Université de Liège, Liège, Belgique, 10 2014.

[115] G. Belov, S. Esler, D. Fernando, P. L. Bodic, and G. L. Nemhauser, "Estimating the size of search trees by sampling with domain knowledge," in *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, 2017, pp. 473–479.

[116] Z. Xing, J. Pei, and E. Keogh, "A brief survey on sequence classification," *ACM SIGKDD Explorations Newsletter*, vol. 12, no. 1, pp. 40–48, Nov. 2010.

[117] E. Klotz and A. M. Newman, "Practical guidelines for solving difficult mixed integer linear programs," *Surveys in Operations Research and Management Science*, vol. 18, no. 1, pp. 18–32, 2013.

[118] M. Deshpande and G. Karypis, "Evaluation of techniques for classifying biological sequences," in *Advances in Knowledge Discovery and Data Mining*, M.-S. Chen, P. S. Yu, and B. Liu, Eds.   Springer Berlin Heidelberg, 2002, pp. 417–431.

[119] T. Lane and C. E. Brodley, "Temporal sequence learning and data reduction for anomaly detection," *ACM Transactions on Information and System Security*, vol. 2, no. 3, pp. 295–331, Aug. 1999.

[120] F. Sebastiani, "Machine learning in automated text categorization," *ACM Computing Surveys*, vol. 34, no. 1, pp. 1–47, Mar. 2002.

[121] T. Achterberg, T. Berthold, and G. Hendel, "Rounding and propagation heuristics for mixed integer programming," in *Operations Research Proceedings 2011*, D. Klatte, H.-J. Lüthi, and K. Schmedders, Eds.   Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 71–76.

[122] H. D. Mittelmann, "MILPlib," accessed 2018. [Online]. Available: http://plato.asu.edu/ftp/milp/

[123] R. J. Lipton and R. W. Regan, "Branch and bound—why does it work?" Gödel's Lost Letter and P=NP, December 2012. [Online]. Available: https://rjlipton.wordpress.com/2012/12/19/branch-and-bound-why-does-it-work/

[124] A. Gleixner, M. Bastubbe, L. Eifler, T. Gally, G. Gamrath, R. L. Gottwald, G. Hendel, C. Hojny, T. Koch, M. E. Lübbecke, S. J. Maher, M. Miltenberger, B. Müller, M. E. Pfetsch, C. Puchert, D. Rehfeldt, F. Schlösser, C. Schubert, F. Serrano, Y. Shinano, J. M. Viernickel, M. Walter, F. Wegscheider, J. T. Witt, and J. Witzig, "The SCIP Optimization Suite 6.0," Zuse Institute Berlin, ZIB-Report 18-26, July 2018. [Online]. Available: http://nbn-resolving.de/urn:nbn:de:0297-zib-69361

[125] S. Maher, M. Miltenberger, J. P. Pedroso, D. Rehfeldt, R. Schwarz, and F. Serrano, "PySCIPOpt: Mathematical programming in Python with the SCIP Optimization Suite," in *Mathematical Software – ICMS 2016*, G.-M. Greuel, T. Koch, P. Paule, and A. Sommese, Eds. Cham: Springer International Publishing, 2016, pp. 301–307.

[126] L. van der Maaten and G. Hinton, "Visualizing high-dimensional data using t-SNE," *Journal of Machine Learning Research*, vol. 9, no. nov, pp. 2579–2605, 2008, pagination: 27.

[127] G. Gamrath and C. Schubert, "Measuring the impact of branching rules for mixed-integer programming," in *Operations Research Proceedings 2017*, N. Kliewer, J. F. Ehmke, and R. Borndörfer, Eds. Cham: Springer International Publishing, 2018, pp. 165–170.

[128] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *International Conference on Learning Representations (ICLR)*, 2015.

[129] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d' Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035.

[130] C. Hansknecht, I. Joormann, and S. Stiller, "Cuts, primal heuristics, and learning to branch for the time-dependent traveling salesman problem," *arXiv preprint 1805.01415*, 2018.

[131] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[132] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *arXiv preprint 1412.3555*, 2014.

[133] A. V. Makkuva, S. Oh, S. Kannan, and P. Viswanath, "Learning in gated neural networks," *CoRR*, vol. abs/1906.02777, 2019.

[134] A. Shrivastava, R. Sukthankar, J. Malik, and A. Gupta, "Beyond skip connections: Top-down modulation for object detection," *CoRR*, vol. abs/1612.06851, 2016.

[135] T. Lin, P. Dollár, R. B. Girshick, K. He, B. Hariharan, and S. J. Belongie, "Feature pyramid networks for object detection," *CoRR*, vol. abs/1612.03144, 2016.

[136] J. Son and A. K. Mishra, "Exgate: Externally controlled gating for feature-based attention in artificial neural networks," *CoRR*, vol. abs/1811.03403, 2018.

[137] Y. Chandak, G. Theocharous, C. Nota, and P. S. Thomas, "Lifelong learning with a changing action set," *CoRR*, vol. abs/1906.01770, 2019.

[138] G. Dulac-Arnold, R. Evans, P. Sunehag, and B. Coppin, "Reinforcement learning in large discrete action spaces," *CoRR*, vol. abs/1512.07679, 2015.

[139] Y. Chandak, G. Theocharous, J. Kostas, S. Jordan, and P. Thomas, "Learning action representations for reinforcement learning," in *Proceedings of the 36th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, K. Chaudhuri and R. Salakhutdinov, Eds., vol. 97. PMLR, 2019, pp. 941–950.

[140] J. Deng, W. Dong, R. Socher, L. jia Li, K. Li, and L. Fei-fei, "ImageNet: A large-scale hierarchical image database," in *CVPR09*, 2009.

[141] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "OpenAI Gym," *arXiv preprint 1606.01540*, 2016.

[142] *Ecole*, 2020. [Online]. Available: https://github.com/ds4dm/ecole

[143] D. Bienstock, "Computational study of a family of mixed-integer quadratic programming problems," *Mathematical Programming*, vol. 74, no. 2, pp. 121–140, 1996.

[144] P. Bonami and M. A. Lejeune, "An exact solution approach for portfolio optimization problems under stochastic and integer constraints," *Operations Research*, vol. 57, no. 3, pp. 650–670, 2009.

[145] O. K. Gupta and A. Ravindran, "Branch and bound experiments in convex nonlinear integer programming," *Management Science*, vol. 31, no. 12, pp. 1533–1546, 1985.

[146] P. Bonami, L. T. Biegler, A. R. Conn, G. Cornuéjols, I. E. Grossmann, C. D. Laird, J. Lee, A. Lodi, F. Margot, N. Sawaya *et al.*, "An algorithmic framework for convex mixed integer nonlinear programs," *Discrete Optimization*, vol. 5, no. 2, pp. 186–204, 2008.

[147] M. A. Duran and I. E. Grossmann, "An outer-approximation algorithm for a class of mixed-integer nonlinear programs," *Mathematical Programming*, vol. 36, no. 3, pp. 307–339, 1986.

[148] P. Belotti, C. Kirches, S. Leyffer, J. Linderoth, J. Luedtke, and A. Mahajan, "Mixed-integer nonlinear optimization," *Acta Numerica*, vol. 22, pp. 1–131, 2013.

[149] MathWorks. (2016) Matlab. [Online]. Available: https://www.mathworks.com/products/matlab.html

[150] J. Puchinger, G. R. Raidl, and U. Pferschy, "The multidimensional knapsack problem: Structure and algorithms," *INFORMS Journal on Computing*, vol. 22, no. 2, pp. 250–265, 2010.

[151] J. H. Friedman, "Stochastic gradient boosting," *Computational Statistics & Data Analysis*, vol. 38, no. 4, pp. 367–378, 2002.

[152] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proceedings of the 32nd International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, F. Bach and D. Blei, Eds., vol. 37.   PMLR, 2015, pp. 448–456.

[153] Y. Wu and K. He, "Group normalization," in *The European Conference on Computer Vision (ECCV)*, 2018, pp. 3–19.

APPENDIX A   LEARNING A CLASSIFICATION OF MIXED-INTEGER QUADRATIC PROGRAMMING PROBLEMS

Authors: Pierre Bonami, Andrea Lodi and Giulia Zarpellon

Published in *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, CPAIOR 2018.[1]

Abstract   Within state-of-the-art solvers such as IBM-CPLEX, the ability to solve both convex and nonconvex Mixed-Integer Quadratic Programming (MIQP) problems to proven optimality goes back few years, yet presents unclear aspects. We are interested in understanding whether for solving a MIQP it is favorable to linearize its quadratic part or not. Our approach exploits machine learning techniques to learn a classifier that predicts, for a given instance, the most suitable resolution method within CPLEX's framework. We aim as well at gaining first methodological insights about the instances' features leading this discrimination. We examine a new dataset and discuss different scenarios to integrate learning and optimization. By defining novel measures, we interpret and evaluate learning results from the optimization point of view.

A.1   Introduction

The tight integration of discrete optimization and machine learning (ML) is a recent but already fruitful research theme: while ML algorithms could profit of choices of discrete type, until now disregarded, various are the discrete optimization settings and situations that could benefit from a ML-based heuristic approach. Although a number of fresh applications is recently appearing in this latter direction, that one could call "learning for optimization" (e.g., [44], [43]), two main topics in this thread of research involve ML-based approaches for the branch-and-bound scheme in Mixed-Integer Linear Programming (MILP) problems (see [25] for a survey on the theme) and the usage of predictions to deal with the solvers' computational aspects and configuration (see, e.g., [34], [35]). We shift from those two main ideas and position ourselves somehow in between, to tackle a new application of ML in discrete optimization. We consider Mixed-Integer Quadratic Programming (MIQP) problems, which prove to be interesting for modeling diverse practical applications (e.g., [143], [144]) as well as a theoretical ground for a first extension of MILP algorithms into nonlinear ones.

Within state-of-the-art solvers such as IBM-CPLEX [28], the ability to solve both convex and

---

[1] Available at [26].

nonconvex MIQPs to proven optimality goes back few years (see, e.g., [102]), but theoretical and computational implications of the employed resolution methods do not seem fully understood yet. We are interested in learning whether it is favorable to linearize the quadratic part of a MIQP or not. As was firstly suggested in [103], we believe that MIQPs should be solved in an intelligent and adapted way in order to improve their resolution process; currently, the decision *linearize vs. not linearize* can be specified by CPLEX users via the linearization switch parameter. We interpret the question *linearize vs. not linearize* as a classification one: we learn a classifier predicting, for a given MIQP, the most suited resolution method within CPLEX, possibly gaining first methodological insights about the problem's features leading to such prediction.

After a quick dive into the MIQPs algorithmic framework (Section A.2), we motivate and state our research question (Section A.3). Methodological details and learning-related aspects are presented in Section A.4, while Section A.5 is devoted to discuss results, new evaluation measures and different scenarios to integrate the learning and the optimization processes.

A.2   Solving MIQPs with CPLEX

We consider general MIQP problems of the form

$$\min \left\{ \frac{1}{2} x^T Q x + c^T x : Ax = b, \ l \leq x \leq u, \ x_j \in \{0, 1\} \ \forall j \in I \right\} \tag{A.1}$$

where $Q = \{q_{ij}\}_{i,j=1,\dots,n} \in \mathbb{R}^{n \times n}$ is a symmetric matrix, $c \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$. Variables $x \in \mathbb{R}^n$ are bounded, and $I \subseteq N = \{1, \dots, n\}$ is the set of indices of variables that are required to be binary. We say that a problem is *pure (binary)* when $I = N$, and *mixed* otherwise; we do not consider the continuous case of $I = \emptyset$. We refer to a MIQP *relaxation* as to the continuous version of (A.1), where integrality requirements are dropped.

Depending on its relaxation being convex or nonconvex, and on the types of variables involved, a MIQP can be tackled in different ways by CPLEX.

Convex problems   A relaxed MIQP is *convex* if and only if the matrix Q is positive semi-definite ($Q \succeq 0$). In this setting, both pure and mixed MIQPs can be solved by the nonlinear programming-based branch and bound [145] (NLP B&B) (see also [146]), a natural extension of the integer linear B&B scheme [3] in which a QP is solved at each node of the tree. Another common resolution approach for convex problems is that of Outer Approximation algorithms [147], which are however not implemented in CPLEX for MIQPs.

Nonconvex problems   When the relaxed MIQP is *not* convex ($Q \not\succeq 0$), variable types play an import role. A binary nonconvex MIQP can be transformed into a convex one by means of augmenting the main diagonal of Q: using $x_j = x_j^2$ for $x_j \in \{0, 1\}$, $x^T Q x$ can be replaced by $x^T (Q + \rho \mathbb{I}_n) x - \rho e^T x$, where $Q + \rho \mathbb{I}_n \succeq 0$ for some suitable $\rho > 0$, $\mathbb{I}_n$ denotes the $n \times n$ identity matrix, and e the vector with all ones. Alternatively, a binary nonconvex MIQP can be linearized and transformed into a MILP. Without performing diagonal augmentation, nonzero terms $q_{ii} x_i^2$ are rewritten as $q_{ii} x_i$, while bilinear terms $q_{ij} x_i x_j$ are handled by the so-called McCormick inequalities [107]: a variable $y_{ij} \geq 0$ is added to represent $x_i x_j$, together with linear constraints

$$x_i + x_j - 1 \leq y_{ij} \text{ if } q_{ij} > 0, \text{ or } y_{ij} \leq x_i, y_{ij} \leq x_j \text{ if } q_{ij} < 0. \tag{A.2}$$

In this way, the problem formulation grows in size, but the resulting model can be solved with standard MILP techniques.

For mixed nonconvex MIQPs, there is no straightforward way to convexify or linearize an instance, and CPLEX relies on the so-called Spatial B&B (see, e.g., [148]) to solve these problems to global optimality.

Although a number of possibilities can be explored to perform linearization and convexification, their discussion is not within the scope of the present paper. For more details, we refer the reader to [102] and the references therein.

A.3   Linearize vs. not linearize

One could assume that the linearization approach discussed for pure nonconvex MIQPs could be beneficial for the convex case as well: a binary problem would turn into a MILP, while in the mixed case one could linearize all bilinear products between a binary and a bounded continuous variable with generalized McCormick inequalities. However, nonzero products between two continuous variables would remain in the formulation, so that a mixed convex MIQP could still be quadratic after linearization, and hence solved with a NLP B&B.

We restrict our focus to *pure convex*, *mixed convex* and *pure nonconvex* problems; the *mixed nonconvex* case should be treated separately, due to the very different setup of Spatial B&B. Currently, in our three cases of interest, the solver provides the user the possibility to switch the linearization on or off by means of the preprocessing parameter `qtolin`, and the default strategy employed by CPLEX is to always perform linearization, although this approach does not dominate in theory the non-linearization one [102].

We aim at learning an offline classifier predicting the most suited resolution approach in a

flexible and instance-specific way. We summarize in what follows the main steps undertaken in the development of our method, leaving the details for the next section.

1. **Dataset generation**: we implement a generator of MIQP instances, spanning across various combinations of structural and optimization parameters.

2. **Features design**: we identify a set of features describing a MIQP in its mathematical formulation and computational behavior.

3. **Labels definition**: we define rigorous procedures to discard troublesome instances, and assess a label depending on running times.

4. **Learning experiments**: we train and test traditional classifiers and interpretable algorithms such as ensemble methods based on Decision Trees.

A.4   Methodological details

We now go through the development steps sketched above more in details, discussing how the dataset is generated, and features and labels defined.

A.4.1   Dataset generation

To build complete MIQP instances, data generation is made of two separate steps. First, symmetric matrices Q are generated by the MATLAB function `sprandsym` [149], to which desired size n, density d and eigenvalues $\lambda_i, i = 1, \ldots, n$ are specified; (in)definiteness of Q is controlled by the spectrum. The second step is implemented with CPLEX Python API: quadratic data can be complemented with a linear vector c, uniform with density d; binary and continuous variables are added in various proportions; finally, a constraints set is defined. We monitor the addition of the following types of constraints, in different combinations:

- a single cardinality constraint $0 \leq \sum_{j \in I} x_j \leq r$, with $r < |I|$ varying;

- a standard simplex constraint $\sum_{j \notin I} x_j = 1, x_j \geq 0$;

- a set of η multi-dimensional knapsack constraints $\sum_{j \in I} w_{ij} x_j \leq f_i$, for $i = 1, \ldots, \eta$. We follow the procedure described in [150] to generate coefficients w_{ij} and f_i, without correlating them to the objective function.

A.4.2 Features design

A (raw) formulation like (A.1) cannot be fed directly as input to a learning algorithm. We depict a MIQP by means of a set of 21 hand-crafted features, summarized in Table A.1. *Static features* describe the instance in terms of variables, constraints and objective function, and are extracted before any solving (pre)process. Few *dynamic features* collect information on the early optimization stages, after the preprocessing and the resolution of the root node relaxation.

A.4.3 Labels definition

To each MIQP we assign a label among L (*linearize*, i.e., qtolin on), NL (*not linearize*, i.e., qtolin off) and T, the latter to account for *tie* cases between L and NL. To deal with performance variability [17], each instance is run in both qtolin modes with 5 different random seeds; we enforce a timelimit of 1h for each run. To monitor troublesome instances, we implement:

- solvability check: instances that cannot be solved within timelimit by any method (neither L nor NL) for any seed are discarded;

- seed consistency check: for each seed, unstable instances with respect to lower and upper bounds of L and NL are discarded;

- global consistency check: a global check on the best upper and lower bounds for the two methods is performed to discard further unstable instances.

If a MIQP passes all these checks, we assign a label. When one mode is never able to solve an instance, the other wins. If both L and NL could solve the instance at least once, running times on each seed are compared and a "seed win" is assigned to one mode if at least 10% better. We assign L or NL only if their seed wins are consistent through the 5 runs, opting for a tie T otherwise.

A.5 Data, experiments and results

The generation procedure is run with MATLAB 9.1.0, Python 2.7 and CPLEX 12.6.3 on a Linux machine, Intel Xeon E5-2637 v4, 3.50 GHz, 16 threads and 128 GB. To label the dataset, we used a grid of 26 machines Intel Xeon X5675, 3.07 GHz (12 threads each) and 96 GB; each problem is restricted to one thread.

We generate 2640 different MIQPs, with size $n \in \{25, 50, \ldots, 200\}$ and density of Q $d \in \{0.2, 0.4, \ldots, 1\}$. For mixed convex MIQPs, the percentage of continuous variables is chosen from $\{0, 20, \ldots, 80\}$. We discard 340 instances due to solvability or consistency failures, ending up with a dataset $\mathcal{D}$ of 2300 problems.

We report in Table A.2 the composition of dataset $\mathcal{D}$ with respect to problem types and assigned labels. The dataset is highly unbalanced: the majority of instances is tagged as NL, with a very small share of T. Also, the NL answer is strongly predominant for mixed convex instances, suggesting that there could be a clear winner method depending on the type of problem itself.

A.5.1 Learning experiments and results

Learning experiments are implemented in Python with Scikit-learn [63], and run on a personal computer with Intel Core i5, 2.3 GHz and 8 GB of memory. We randomly split $\mathcal{D}$ into $\mathcal{D}_{\text{train}}$ and $\mathcal{D}_{\text{test}}$, a training and a test sets of, respectively, 1725 (75%) and 575 (25%) instances; data is normalized in $[-1, 1]$. We perform training with 5-fold cross validation to grid-search hyper-parameters, and test on the neutral $\mathcal{D}_{\text{test}}$. We try Support Vector Machine (SVM) with RBF kernel [112], together with Random Forests (RF) [113], Extremely Randomized Trees (EXT) [88] and Gradient Tree Boosting (GB) [151].

Our first experiment involves a multiclass scheme with labels $\{\mathsf{L}, \mathsf{NL}, \mathsf{T}\}$, and exploits all features. Table A.4a reports the standard measures for classification in this setting: for all classifiers we compare accuracy, precision, recall, f1-score (weighted by classes' supports, to account for unbalance). In this setting, RF is best performing in all measures. Features importance scores among RF, EXT and GB show that the subset of features that are more influential for the prediction comprises both dynamic features (difference of lower bounds and times at root node) and information on the convexity of the problem (e.g., value of the smallest nonzero eigenvalue and spectral norm of Q).

Examining the classifiers' confusion matrices, a major difficulty seems to be posed by the T class, which is (almost) always misclassified. Ultimately, we aim at providing a reliable classification of those "extreme" cases for which a change in the resolution approach produces a change in the instance being solved or not. Thus, we carry out further experiments in a binary setting: we remove all tie cases and rescale the data accordingly. All measures are overall improved for the new binary classifiers, and again RF performs as the best algorithm.

We also try classifiers trained without dynamic features: albeit this may sound in conflict with the features importance scores mentioned above, from the optimization solver's point

Table A.1 Overview and brief description of the complete features set.

#	Group name	Features general description
	Static features	
2	Generic problem type	Size of the problem, variables proportions per type.
2	Constraints matrix composition	Density w.r.t. different types of variables, magnitudes of nonzero (nnz) coefficients.
5	Quadratic matrix composition	Magnitudes of coefficients, nnz diagonal and bilinear (*continuous · continuous*) and (*binary · continuous*).
7	Spectrum description	Shares of positive/negative eigenvalues, magnitude and value of the smallest, trace and spectral norm.
3	Other properties of Q	Density, rank, a measure of "diagonal dominance".
	Dynamic features	
2	Root node information	Difference of lower bounds and resolution times at the root node, between *linearize* and *not linearize.*

Table A.2 Composition of dataset $\mathcal{D}$. For each type and label we report the total number of instances and their percentage.

	L	NL	T	**Total** (%)
0-1 convex	195	600	35	830 (**0.36**)
0-1 nonconvex	392	312	39	743 (**0.32**)
mixed convex	11	701	15	727 (**0.32**)
Total (%)	598 (**0.26**)	1613 (**0.70**)	89 (**0.04**)	**2300**

of view is it useful to test a scenario in which a prediction is cast without the need of solving twice the root MIQP. All measures slightly deteriorate without dynamic features, and SVM becomes the best performing algorithm; nonetheless, "static" predictors and their original counterparts are coherent in their (mis)classifications on $\mathcal{D}_{\text{test}}$.

Results in a setting simplified in terms of both labels and features are reported in Table A.4b: performance is balanced in the improvement brought by the removal of ties, and the degradation due to the absence of dynamic features, and again SVM performs better.

A.5.2 Complementary optimization measures

To determine the effectiveness of our learned approach with respect to the solver's strategy, we define "optimization measures" scoring and evaluating the classifiers in terms of resolution runtimes.

We run each instance i of $\mathcal{D}_{\text{test}}$ for three `qtolin` values - CPLEX default (`DEF`), `L` and

Table A.3 Classification measures for different learning settings. The best performing classifiers are boldfaced.

	SVM	**RF**	EXT	GB
Accuracy	85.22	**88.87**	84.00	87.65
Precision	81.91	**85.51**	81.26	84.79
Recall	85.22	**88.87**	84.00	87.65
F1-score	83.16	**87.11**	82.52	86.19

(a) Multiclass - All features

	SVM	RF	EXT	GB
Accuracy	**86.80**	86.08	85.53	86.62
Precision	**86.48**	85.69	85.20	86.32
Recall	**86.80**	86.08	85.53	86.62
F1-score	**86.28**	85.53	85.30	86.03

(b) Binary - Static features

NL. Each problem is run only once, with timelimit of 1h; we focus on the Multiclass and All features setting. We remove never-solved instances, to remain with 529 problems in $\mathcal{D}_{\text{test}}$. For each classifier clf, we associate to the vector of its predicted labels y_{clf} a vector of predicted times t_{clf} by selecting t_{L}^i or t_{NL}^i depending on y_{clf}^i for $i \in \mathcal{D}_{\text{test}}$ (we choose their average if a tie was predicted). We also build t_{best} (t_{worst}) selecting runtimes of the correct (wrong) labels for the samples. Note that t_{DEF} is directly available, without labels' vector.

Sum of predicted runtimes We compare $\sigma_{\text{clf}} := \sum_{i \in \mathcal{D}_{\text{test}}} t_{\text{clf}}^i$ for clf $\in$ {SVM, RF, EXT, GB} with σ_{best}, σ_{worst} and σ_{DEF}. Results are in Table A.5a: RF is the closest to *best* and the farthest from *worst*; also, DEF could take up to 4x more time to run MIQPs in $\mathcal{D}_{\text{test}}$ compared to a trained classifier. Note that the real gain in time could be even bigger than this, given the fact that we set a timelimit of 1h.

Normalized time score We then consider the shifted geometric mean of t_{clf} over $\mathcal{D}_{\text{test}}$, normalized between *best* and *worst* cases to get a score $N\sigma_{\text{clf}} \in [0, 1]$:

$$sgm_{\text{clf}} := \sqrt[|\mathcal{D}_{\text{test}}|]{\prod_{i \in \mathcal{D}_{\text{test}}} (t_{\text{clf}}^i + 0.01)} - 0.01, \quad N\sigma_{\text{clf}} := \frac{sgm_{\text{worst}} - sgm_{\text{clf}}}{sgm_{\text{worst}} - sgm_{\text{best}}}. \tag{A.3}$$

The measure is reported in Table A.5a: all predictors are very close to 1 (this score highly reflects classification performance), while DEF is almost halfway between *best* and *worst*.

The presence of timelimiting cases in the test runs is also well reflected in σ_{clf} and $N\sigma_{\text{clf}}$, which are better for classifiers hitting timelimits less frequently (3 times only for RF, 39 for DEF). Note that both L and NL do reach the limit without finding a solution (38 and 55 times, respectively), and that due to variability even *best* hits the timelimit once. We compute σ_{clf} and $N\sigma_{\text{clf}}$ in the Binary - Static features setting as well. Results in Table A.5b are in line with what previously discussed for this setup.

Table A.4 Complementary optimization measures. Best classifiers are boldfaced.

	SVM	**RF**	EXT	GB	DEF
$\sigma_{\text{clf}}/\sigma_{\text{best}}$	1.49	**1.31**	1.43	1.35	5.77
$\sigma_{\text{worst}}/\sigma_{\text{clf}}$	7.48	**8.49**	7.81	8.23	1.93
$\sigma_{\text{DEF}}/\sigma_{\text{clf}}$	3.88	**4.40**	4.04	4.26	-
$N\sigma_{\text{clf}}$	0.98	**0.99**	0.98	0.99	0.42

(a) Multiclass - All features

	SVM	RF	EXT	GB	DEF
$\sigma_{\text{clf}}/\sigma_{\text{best}}$	**1.80**	2.04	2.01	1.82	5.81
$\sigma_{\text{worst}}/\sigma_{\text{clf}}$	**6.23**	5.50	5.59	6.19	1.93
$\sigma_{\text{DEF}}/\sigma_{\text{clf}}$	**3.22**	2.85	2.89	3.20	-
$N\sigma_{\text{clf}}$	**0.98**	0.98	0.98	0.98	0.43

(b) Binary - Static features

A.6   Conclusions and ongoing research

We propose a learning framework to investigate the question *linearize vs. not linearize* for MIQP problems. Results on a generated dataset are satisfactory in terms of classification performance and promising for their interpretability. Novel scoring measures positively evaluate the classifiers' performance from the optimization point of view, showing significant improvements with respect to CPLEX default strategy in terms of running times.

In ongoing and future research, we plan to focus on four main directions.

- Analyze other benchmark datasets: the analysis of public libraries containing MIQPs (e.g., [24]) is crucial to understand how representative the synthetic $\mathcal{D}$ is of commonly used instances, which can then be used to form a more meaningful and comprehensive final dataset.

  So far, we analyzed a share of CPLEX internal MIQP testbed $\mathcal{C}_{\text{test}}$ of 175 instances: the data is very different from $\mathcal{D}$ in features' distribution ($\mathcal{C}_{\text{test}}$ is dominated by the presence of very structured combinatorial MIQPs, like Max-Cut and Quadratic Assignment Problems). The majority class is that of ties, followed by L and with very few NL cases. Preliminary experiments on $\mathcal{C}_{\text{test}}$ used as a test set for classifiers trained on $\mathcal{D}_{\text{train}}$ produce very poor classification results, as most often misclassification happens in form of a T predicted as NL. In fact, complementary optimization measures are not discouraging: given that $\mathcal{C}_{\text{test}}$ contains mostly ties, albeit the high misclassification rate, the loss in terms of solver's performance is not dramatic.

- Deepen features importance analysis, to get and interpret methodological insights on the reasons behind the decision *linearize vs. not linearize.* As we mentioned in Section A.5, the problem type itself might draw an important line in establishing the winning method, which seems strongly linked to the information collected at the root node.

- Identify the best learning scenario, in order to successfully integrate the learning framework with the solver. We already considered the simplified Binary - Static features one; it could be interesting to perform static features selection based on their correlation with dynamic ones.

- Define a custom loss function: the complementary optimization measures that we propose showed effective in capturing the optimization performance as well as the classification one. We plan to use these and other intuitions to craft a custom loss/scoring function to train/validate the learning algorithm, in a way tailored to the solver's performance on MIQPs.

APPENDIX B   ADDITIONAL MATERIAL – A CLASSIFIER TO DECIDE ON THE LINEARIZATION OF MIQPS IN CPLEX

Table B.1 Description of features in the `Initial` set (60). Features marked with * are part of the `Selected` set.

	Name	Description
	Static features	
	RSizes	Ratio of sizes m/n
*	RBin	Ratio of binary variables, over n
*	RContInt	Ratio of continuous and integer variables, over n
	RNnzDiagBin	Ratio of non-zero (nnz) coefficients in Q diagonal for binary variables
*	RNnzDiagContInt	Ratio of nnz coefficients in Q diagonal for continuous and integer variables
	DiagDensity	Density of diagonal entries of Q
*	OutDiagDensity	Density of non-diagonal entries of Q
*	QDensity	Density of Q
*	RBinBin	Ratio of nnz products between binary variables in Q
*	RContContIntInt	Ratio of nnz products between continuous or integer variables in Q
	RMixedBin	Ratio of mixed-type products involving binaries
	RMixedContInt	Ratio of mixed-type products involving continuous and/or integers
*	RNonLinTerms	Ratio of nnz non-linearizable terms, over n^2
	RNonLinTermsNnz	Ratio of nnz non-linearizable terms, over nnz
*	RelVarsLinInc	Relative size increase of potential linearization, over n
	RelConssLinInc	Relative size increase of potential linearization, over m
*	RLinSizes	Sizes m/n ratio after potential linearization
*	NormMaxDegBin	Maximum connectivity degree in Q among binary variables, over $n-1$
*	NormMaxDegContInt	Maximum connectivity degree in Q among continuous and integer variables, over $n-1$
	AvgDiagDom	Averaged "diagonal dominance" on rows [26]
	RDiagCoeff	Ratio of biggest and smallest diagonal nnz coefficients of Q, in absolute value
	ROutDiagCoeff	Ratio of biggest nnz diagonal coefficient and smallest out diagonal one, in absolute value
	RNnzBinLin	Ratio nnz binary variables in linear term
*	RNnzContIntLin	Ratio nnz continuous and integers variables in linear term
	HasLinearTerm	Boolean, whether there is a linear term
	LinDensity	Density of the linear term
	RLinCoeff	Ratio of biggest on smallest linear coefficients, in absolute value
*	ConssDensity	Density of constraints matrix A
	RConssBin	Ratio of constraints involving binary variables, over m

Table B.1 – continued from previous page

	Name	Description
	RConssCont	Ratio of constraints involving continuous variables, over m
*	RConssInt	Ratio of constraints involving integer variables, over m
	RConssCoeff	Ratio biggest on smallest nnz constraints coefficients, in absolute value
	RRhsCoeff	Ratio magnitudes smallest on biggest nnz rhs coefficients, in absolute value
	RQTrace	Trace of Q, over n
	QSpecNorm	Spectral norm of Q
*	RQRankEig	Rank of Q over n (i.e., ratio of nnz eigenvalues of Q)
*	HardEigenPerc	Portion of problematic (hard) eigenvalues in Q
	AvgSpecWidth	Width of Q spectrum, over n
	RPosEigen	Ratio of positive eigenvalues, over n
	RNegEigen	Ratio of negative eigenvalues, over n
	RZeroEigen	Ratio of zero eigenvalues, over n
	RAbsEigen	Ratio of min and max eigenvalues, in absolute value
	RNZeroEigenDiff	Ratio of difference between original and corrected eigenvalues
	HardEigenPercDiff	Ratio of difference between original and corrected hard eigenvalues

Preprocessing features

	Name	Description
*	prep_RelVarsIncL	Relative variables increase after L preprocessing
	prep_RelVarsIncNL	Relative variables increase after NL preprocessing
*	prep_RelConssIncL	Relative constraints increase after L preprocessing
	prep_RelConssIncNL	Relative constraints increase after NL preprocessing
*	prep_RSizesL	Sizes m/n ratio after L preprocessing
	prep_RSizesNL	Sizes m/n ratio after NL preprocessing
*	prep_ConssDensityL	Density of constraints matrix after L preprocessing
	prep_ConssDensityNL	Density of constraints matrix after NL preprocessing
	prep_ConssDensityDiff	Difference of density of constraints after preprocessing, between L and NL
	prep_RelConssDensityL	Relative density of constraints after L preprocessing with respect to original one
	prep_RelConssDensityNL	Relative density of constraints after NL preprocessing with respect to original one

Root node features

	Name	Description
	root_RtTimeDiff	Difference of total root times (comprising preprocessing), between L and NL
	root_RLPTimeDiff	Difference of LP root times, between L and NL
	root_SignRDBDiff	Sign of dual bounds at root (1 if L better, -1 if NL better)
	root_RelRDBDiff	Relative difference of bounds at root
	root_RelSignRDBDiff	Signed relative difference of L and NL bounds at root

Table B.2 Top-10 features identified by `RF` importance scores, in the multi-class and binary setting with `Initial` and `Selected` features. The reported scores are averages across the five `RF` models trained in each configuration.

Rank	Score	Feature
1.	0.1826	`root_RtTimeDiff`
2.	0.0897	`prep_ConssDensityL`
3.	0.0862	`prep_RelVarsIncL`
4.	0.0779	`root_RelRDBDiff`
5.	0.0512	`root_RelSignRDBDiff`
6.	0.0407	`prep_RSizesL`
7.	0.0391	`prep_RelConssDensityL`
8.	0.039	`root_RLPTimeDiff`
9.	0.0311	`QDensity`
10.	0.0293	`OutDiagDensity`

(a) `MultiLabel` - `Initial`

Rank	Score	Feature
1.	0.0762	`QDensity`
2.	0.07	`RBinBin`
3.	0.068	`root_RtTimeDiff`
4.	0.065	`OutDiagDensity`
5.	0.0545	`root_RelSignRDBDiff`
6.	0.0419	`NormMaxDegBin`
7.	0.0327	`prep_ConssDensityL`
8.	0.0314	`prep_RelVarsIncL`
9.	0.0257	`prep_RelConssDensityL`
10.	0.0252	`root_RelRDBDiff`

(b) `BinLabel` - `Initial`

Rank	Score	Feature
1.	0.203	`prep_RelVarsIncL`
2.	0.1807	`prep_ConssDensityL`
3.	0.1113	`prep_RSizesL`
4.	0.0571	`QDensity`
5.	0.055	`RelVarsLinInc`
6.	0.0519	`prep_RelConssIncL`
7.	0.0519	`OutDiagDensity`
8.	0.0513	`RBinBin`
9.	0.0441	`NormMaxDegBin`
10.	0.0306	`RQRankEig`

(c) `MultiLabel` - `Selected`

Rank	Score	Feature
1.	0.1335	`QDensity`
2.	0.1177	`OutDiagDensity`
3.	0.1026	`RBinBin`
4.	0.0774	`prep_RelVarsIncL`
5.	0.067	`NormMaxDegBin`
6.	0.0633	`prep_ConssDensityL`
7.	0.0582	`RelVarsLinInc`
8.	0.0559	`prep_RSizesL`
9.	0.054	`RQRankEig`
10.	0.0453	`RNonLinTerms`

(d) `BinLabel` - `Selected`

APPENDIX C   ADDITIONAL MATERIAL – PARAMETERIZING B&B SEARCH TREES TO LEARN BRANCHING POLICIES

Dataset curation

To curate a dataset of heterogeneous MILP instances, we consider the standard benchmark libraries MIPLIB 3, 2010 and 2017[1] [16, 23, 81], together with the collection of [122]. We assess the problems by analyzing B&B roll-outs of SCIP with its default branching rule (`relpscost`) and a `random` one, enforcing a time limit of 1h in the same solver setting used for our experiments (see later). We focus on instances whose tree exploration is on average relatively contained (in the tens/hundreds of thousands nodes, maximum) and whose optimal value is known. This choice is primarily motivated by the need of ensuring a fair comparison among branching policies in terms of tree size, which is more easily achieved when roll-outs do not hit the time-limit. We also remove problems that are solved at the root node (i.e., those for which no branching was performed).

Final training and test sets comprise 19 and 8 instances, respectively, for a total of 27 problems. They are summarized in Table C.1, where we report their size, the number of binary/integer/continuous variables, the number of constraints, their membership in the train/test split and their library of origin. The constraints of each problem are of different types and give rise to various structures.

Hand-crafted input features

Hand-crafted input features for candidate variables (C_t) and tree state (*Tree$_t$*) are reported in Table C.4. To ease their reading, we present them subdivided in groups, and synthetically describe them by the SCIP API functions with which they are computed. We make use of different functions to normalize and compare the solver inputs.

To compute the branching scores s_i of a candidate variable $i \in \mathcal{C}_t$, with respect to an average score s_{avg}, we use the formula implemented in SCIP `relpscost`[2]:

$$\texttt{varScore}(s_i, s_{avg}) = 1 - \left(\frac{1}{1 + s_i / \max\{s_{avg}, 0.1\}} \right). \tag{C.1}$$

---

[1] http://miplib2017.zib.de

[2] https://scip.zib.de/doc-6.0.0/html/branch__relpscost_8c_source.php#l00524

Table C.1 The curated MILP dataset. For each instance we report: the number of variables (Vars) and their types (binary, integers and continuous), the number of constraints (Conss), the membership in the train/test split and the library of origin.

Name	Vars	Types (bin - int - cont)	Conss	Set	Library
air04	8904	8904 - 0 - 0	823	train	MIPLIB 3
air05	7195	7195 - 0 - 0	426	train	MIPLIB 3
dcmulti	548	75 - 0 - 473	290	train	MIPLIB 3
eil33-2	4516	4516 - 0 - 0	32	train	MIPLIB 2010
istanbul-no-cutoff	5282	30 - 0 - 5252	20346	train	MIPLIB 2017
l152lav	1989	1989 - 0 - 0	97	train	MIPLIB 3
lseu	89	89 - 0 - 0	28	train	MIPLIB 3
misc03	160	159 - 0 - 1	96	train	MIPLIB 3
neos20	1165	937 - 30 - 198	2446	train	MILPLib
neos21	614	613 - 0 - 1	1085	train	MILPLib
neos-476283	11915	5588 - 0 - 6327	10015	train	MIPLIB 2010
neos648910	814	748 - 0 - 66	1491	train	MILPLib
pp08aCUTS	240	64 - 0 - 176	246	train	MIPLIB 3
rmatr100-p10	7359	100 - 0 - 7259	7260	train	MIPLIB 2010
rmatr100-p5	8784	100 - 0 - 8684	8685	train	MIPLIB 2010
sp150x300d	600	300 - 0 - 300	450	train	MIPLIB 2017
stein27	27	27 - 0 - 0	118	train	MIPLIB 3
swath1	6805	2306 - 0 - 4499	884	train	MIPLIB 2017
vpm2	378	168 - 0 - 210	234	train	MIPLIB 3
map18	164547	146 - 0 - 164401	328818	test	MIPLIB 2010
mine-166-5	830	830 - 0 - 0	8429	test	MIPLIB 2010
neos11	1220	900 - 0 - 320	2706	test	MILPLib
neos18	3312	3312 - 0 - 0	11402	test	MIPLIB 2010
ns1830653	1629	1458 - 0 - 171	2932	test	MIPLIB 2010
nu25-pr12	5868	5832 - 36 - 0	2313	test	MIPLIB 2017
rail507	63019	63009 - 0 - 10	509	test	MIPLIB 2010
seymour1	1372	451 - 0 - 921	4944	test	MIPLIB 2017

As in [5], we normalize inputs that naturally span different ranges by the following:

$$\texttt{gNormMax}(x) = \max\left\{\frac{x}{x+1}, 0.1\right\}. \tag{C.2}$$

To compare commensurable quantities (e.g., upper and lower bounds), we compute measures of relative distance and relative position:

$$\texttt{relDist}(x,y) = \begin{cases} 0 & \text{, if } xy < 0 \\ \frac{|x-y|}{\max\{|x|,|y|,1e-10\}} & \text{, else} \end{cases} \tag{C.3}$$

$$\texttt{relPos}(z,x,y) = \frac{|x-z|}{|x-y|}. \tag{C.4}$$

We also make use of usual statistical functions such as `min`, `max`, `mean`, standard deviation `std` and 25-75% quantile values (denoted in Table C.4 as `q1` and `q3`, respectively).

Further information on each feature can be gathered by searching the SCIP online documentation at `https://scip.zib.de/doc-6.0.1/html/`.

Solver setting and hardware

Regarding the MILP solver parametric setting, we use SCIP 6.0.1 and set a time-limit of 1h on all B&B evaluations. We leave on presolve routines and cuts separation (as in default mode), while disabling *all* primal heuristics and reoptimization (also off at default). To control SB side-effects and properly compute the *fair* number of nodes [127], we additionally turn off SB conflict analysis and the use of probing bounds identified during SB evaluations. We also disable feasibility checking of LP solutions found during SB with propagation, and always trigger the reevaluation of SB values. Finally, the known optimal solution value is provided as cutoff to each model, and a random seed determines variables' permutations. Parameters are summarized in Table C.2.

Table C.2 SCIP parametric setting.

limits/time = 3600
presolving/maxrounds = -1
separating/maxrounds = -1
separating/maxroundsroot = -1
heuristics/*/freq = -1
reoptimization/enable = False
conflict/usesb = False
branching/fullstrong/probingbounds = False
branching/relpscost/probingbounds = False
branching/checksol = False
branching/fullstrong/reevalage = 0
model.setObjlimit(cutoff_value)
randomization/permutevars = True
randomization/permutationseed = scip_seed

For the IL experiments, we used the following hardware: Two Intel Core(TM) i7-6850K CPU @ 3.60GHz, 16GB RAM and an NVIDIA TITAN Xp 12GB GPU. Evaluations of SCIP branching rules ran on dual Intel(R) Xeon(R) Gold 6142 CPU @ 2.60GHz, equipped with 512GB of RAM.

Table C.3 Variability scores *VS* are reported for `relpscost`, and are computed using the 5 runs with $k = 0$ (i.e., SCIP default runs, over seeds $\{0, \ldots, 4\}$). Total number of nodes explored by data collection runs with k random branchings, in shifted geometric means over 5 runs is also reported. Finally, VS_k is the coefficient of variation of the five means, across different k's.

Instance	Set	VS	$k = 0$	$k = 1$	$k = 5$	$k = 10$	$k = 15$	VS_k
air04	train	0.20	8.19	12.02	46.57	85.35	119.49	0.79
air05	train	0.26	60.25	61.07	115.94	196.27	274.44	0.59
dcmulti	train	0.21	9.38	13.75	27.53	34.99	45.00	0.50
eil33-2	train	0.69	583.34	648.47	492.95	531.37	441.24	0.13
istanbul-no-cutoff	train	0.11	242.39	234.01	271.35	279.38	311.39	0.10
l152lav	train	0.36	10.14	16.54	29.31	55.51	61.14	0.59
lseu	train	0.43	148.99	152.65	154.16	182.55	177.35	0.09
misc03	train	0.38	12.11	10.59	13.80	22.59	31.80	0.44
neos20	train	1.22	200.26	282.68	557.15	434.03	944.75	0.54
neos21	train	0.15	668.44	771.77	898.79	1110.82	1158.07	0.21
neos648910	train	0.60	39.83	48.16	65.84	41.05	59.72	0.20
neos-476283	train	0.48	204.88	219.58	384.86	480.37	715.78	0.47
pp08aCUTS	train	0.31	69.66	80.39	92.60	69.94	76.43	0.11
rmatr100-p5	train	0.04	411.93	419.21	451.01	461.83	494.09	0.07
rmatr100-p10	train	0.03	806.35	799.24	860.60	933.80	965.07	0.08
sp150x300d	train	1.70	182.22	462.45	484.55	483.89	439.69	0.28
stein27	train	0.42	926.82	1062.69	1098.41	1162.57	1154.01	0.08
swath1	train	0.53	298.58	280.49	230.12	256.84	267.55	0.09
vpm2	train	0.19	199.46	180.93	275.57	273.33	316.82	0.20
map18	test	0.09	270.25	309.77	401.79	447.34	489.85	0.21
mine-166-5	test	0.82	175.10	70.77	642.33	942.63	1619.75	0.81
neos11	test	0.30	2618.27	3114.62	3488.40	2898.41	2659.96	0.11
neos18	test	0.53	2439.29	2747.77	4061.40	4655.59	5714.05	0.31
ns1830653	test	0.09	3489.07	3913.58	4091.59	4839.39	4772.73	0.12
nu25-pr12	test	1.18	21.39	16.97	56.04	101.34	119.05	0.66
rail507	test	0.08	543.39	562.09	854.76	1207.15	1196.33	0.33
seymour1	test	0.07	866.32	1174.18	1825.04	2739.45	3313.87	0.47

Data augmentation

To augment our dataset, we (i) run MILP instances with different random seeds to exploit performance variability [17], and (ii) perform k random branchings at the top of the tree, before switching to the default SCIP branching rule and collect data. To quantify the effects of such operations in diversifying the search trees, we compute coefficients of variations of performance measurements [16]. In particular, assuming performance measurements $n_l, l = 1 \ldots L$ are available, we compute the variability score *VS* as

$$VS = \frac{L}{\sum_{l=1}^{L} n_l} \sqrt{\sum_{l=1}^{L} \left(n_l - \frac{\sum_{l=1}^{L} n_l}{L} \right)^2}. \tag{C.5}$$

Table C.3 reports such coefficients for all instances, using as performance measures the number of nodes explored in the five runs from (i). Similarly, we report the shifted geometric means of the number of nodes over the five runs for each $k \in \{0, 1, 5, 10, 15\}$, and additionally compute the variability of those means, across different k's (VS_k).

IL optimization dynamics

Best policies We present additional plots of the optimization dynamics for the best selected NoTree and TreeGate policies. Figure C.1 shows the training loss curves, as well as top-1 and top-5 validation accuracy curves. In general, we see that the TreeGate policy enjoys a better conditioned optimization. Note however that for top-5 validation accuracy the two policies are quite close.
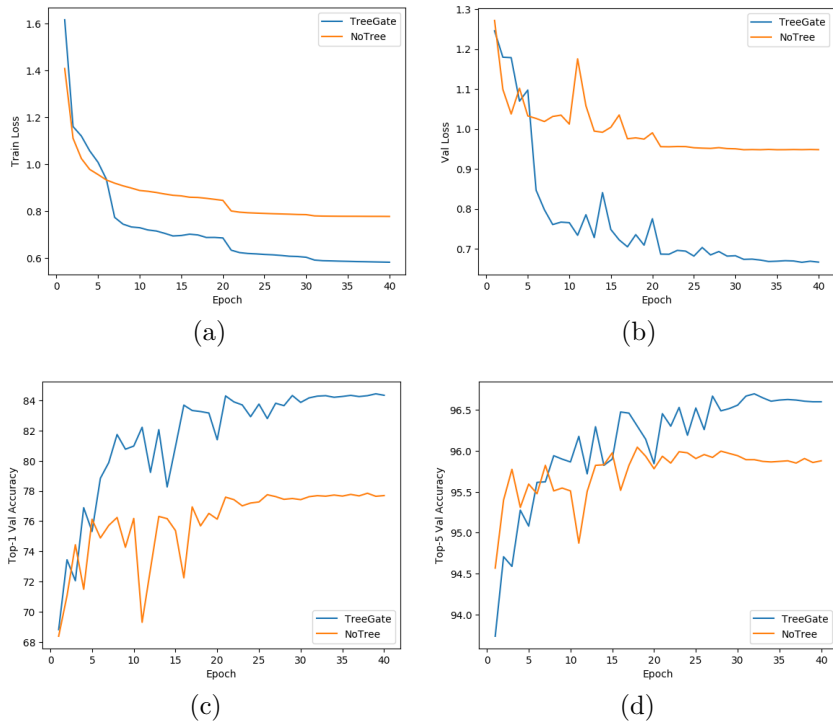

Figure C.1 (a) Train and (b) validation loss curves for the best NoTree (orange) and TreeGate (blue) policies. (c) Validation top-1 and (d) top-5 accuracy plots for the best NoTree and TreeGate policies.

Instability of batch-norm As observed in Figure C.1, optimization dynamics for NoTree seem to be of a much slower nature than those of TreeGate. One common option to speed up training is to use batch normalization (BN) [152]. In our architectures for branching, one

may view the cardinality of the candidate sets $|\mathcal{C}_t|$ as a batch dimension. When learning to branch across heterogeneous MILPs, such batch dimension can (and will) vary by orders of magnitude. Practically, our dataset has $|\mathcal{C}_t|$ varying from < 10 candidates to over 300. To this end, BN has been shown to struggle in the small-batch setting [153], and in general we were unsure of the reliability of BN with such variable batch-sizes.

Indeed, in our initial trials with BN we observed highly unreliable performance. Two troubling outcomes emerge when using BN in our NoTree policies: 1) the validation accuracy varies wildly, as shown in Figure C.2, or 2) the NoTree+BN policy exhibits a stable validation accuracy curve, but would time-limit on *train instances*, i.e., would perform poorly in terms of solver performance. In particular, case 2) happened for a NoTree+BN policy with hidden size $h = 64$ and $LR = 0.001$, reaching the 1h time-limit on train instance neos-476283, over all five runs (on different seeds); the geometric mean of explored nodes was 66170.66. We remark that in our non-BN experiments, all of our trained policies (both TreeGate and NoTree) managed to solve all the train instances without even coming close to time-limiting. Moreover, none of our training and validation curves ever remotely resemble those in Figure C.2b.

For these reasons we opted for a more streamlined presentation of our results, without BN in the current framework. We leave it for future work to analyze the relationship between the nature of local minima in the IL optimization landscape and solver performance.
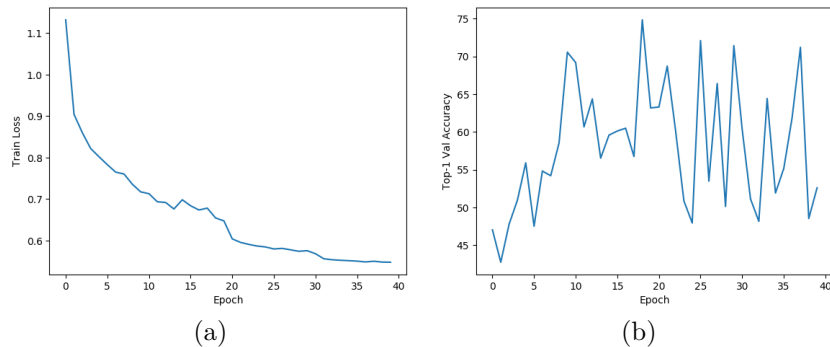

(a)   (b)

Figure C.2 (a) Train loss and (b) validation top-1 Accuracy for NoTree+BN policy with hidden size $h = 64$, $LR = 0.001$.

Table C.4 Hand-crafted input features. Features are doubled [x2] when computed for both upward and downward branching directions. For features about open nodes, open_lbs (resp. open_ds) denotes the list of lower bound estimates (resp. depths) of the open nodes.

Group description (#)	Feature formula (SCIP API)
Candidate state $[C_t]_i, i \in \mathcal{C}_t$	

Table C.4 – continued from previous page

Group description (#)	Feature formula (SCIP API)
General solution (2)	SCIPvarGetLPSol
	SCIPvarGetAvgSol
Branchings depth (2)	1 - (SCIPvarGetAvgBranchdepthCurrentRun / SCIPgetMaxDepth) [x2]
Branching scores (5)	varScore(SCIPgetVarConflictScore, SCIPgetAvgConflictScore)
	varScore(SCIPgetVarConflictlengthScore, SCIPgetAvgConflictlengthScore)
	varScore(SCIPgetVarAvgInferenceScore, SCIPgetAvgInferenceScore)
	varScore(SCIPgetVarAvgCutoffScore, SCIPgetAvgCutoffScore)
	varScore(SCIPgetVarPseudocostScore, SCIPgetAvgPseudocostScore)
PC stats (6)	SCIPgetVarPseudocostCountCurrentRun / SCIPgetPseudocostCount [x2]
	SCIPgetVarPseudocostCountCurrentRun / SCIPvarGetNBranchingsCurrentRun [x2]
	SCIPgetVarPseudocostCountCurrentRun / branch_count [x2]
Implications (2)	SCIPvarGetNImpls [x2]
Cliques (2)	SCIPvarGetNCliques / SCIPgetNCliques [x2]
Cutoffs (2)	gNormMax(SCIPgetVarAvgCutoffsCurrentRun) [x2]
Conflict length (2)	gNormMax(SCIPgetVarAvgConflictlengthCurrentRun) [x2]
Inferences (2)	gNormMax(SCIPgetVarAvgInferencesCurrentRun) [x2]
Search tree state Tree$_t$	
Current node (8)	SCIPnodeGetDepth / SCIPgetMaxDepth
	SCIPgetPlungeDepth / SCIPnodeGetDepth
	relDist(SCIPgetLowerbound, SCIPgetLPObjval)
	relDist(SCIPgetLowerboundRoot, SCIPgetLPObjval)
	relDist(SCIPgetUpperbound, SCIPgetLPObjval)
	relPos(SCIPgetLPObjval, SCIPgetUpperbound, SCIPgetLowerbound)
	len(getLPBranchCands) / getNDiscreteVars
	nboundchgs / SCIPgetNVars
Nodes and leaves (8)	SCIPgetNObjlimLeaves / nleaves
	SCIPgetNInfeasibleLeaves / nleaves
	SCIPgetNFeasibleLeaves / nleaves
	(SCIPgetNInfeasibleLeaves + 1) / (SCIPgetNObjlimLeaves + 1)
	SCIPgetNNodesLeft / SCIPgetNNodes
	nleaves / SCIPgetNNodes
	ninternalnodes / SCIPgetNNodes
	SCIPgetNNodes / ncreatednodes
Depth and backtracks (4)	nactivatednodes / SCIPgetNNodes
	ndeactivatednodes / SCIPgetNNodes
	SCIPgetPlungeDepth / SCIPgetMaxDepth

Table C.4 – continued from previous page

Group description (#)	Feature formula (SCIP API)
LP iterations (4)	SCIPgetNBacktracks / SCIPgetNNodes
	`log`(SCIPgetNLPIterations / SCIPgetNNodes)
	`log`(SCIPgetNLPs / SCIPgetNNodes)
	SCIPgetNNodes / SCIPgetNLPs
	SCIPgetNNodeLPs / SCIPgetNLPs
Gap (4)	`log`(primaldualintegral)
	SCIPgetGap / lastsolgap
	SCIPgetGap / firstsolgap
	lastsolgap / firstsolgap
Bounds and solutions (5)	`relDist`(SCIPgetLowerboundRoot, SCIPgetLowerbound)
	`relDist`(SCIPgetLowerboundRoot, SCIPgetAvgLowerbound)
	`relDist`(SCIPgetUpperbound, SCIPgetLowerbound)
	SCIPisPrimalboundSol
	nnodesbeforefirst / SCIPgetNNodes
Average scores (12)	`gNormMax`(SCIPgetAvgConflictScore)
	`gNormMax`(SCIPgetAvgConflictlengthScore)
	`gNormMax`(SCIPgetAvgInferenceScore)
	`gNormMax`(SCIPgetAvgCutoffScore)
	`gNormMax`(SCIPgetAvgPseudocostScore)
	`gNormMax`(SCIPgetAvgCutoffs) [x2]
	`gNormMax`(SCIPgetAvgInferences) [x2]
	`gNormMax`(SCIPgetPseudocostVariance) [x2]
	`gNormMax`(SCIPgetNConflictConssApplied)
Open nodes bounds (12)	`len`(open_lbs at {`min`, `max`}) / nopen [x2]
	`relDist`(SCIPgetLowerbound, `max`(open_lbs))
	`relDist`(`min`(open_lbs), `max`(open_lbs))
	`relDist`(`min`(open_lbs), SCIPgetUpperbound)
	`relDist`(`max`(open_lbs), SCIPgetUpperbound)
	`relPos`(`mean`(open_lbs), SCIPgetUpperbound, SCIPgetLowerbound)
	`relPos`(`min`(open_lbs), SCIPgetUpperbound, SCIPgetLowerbound)
	`relPos`(`max`(open_lbs), SCIPgetUpperbound, SCIPgetLowerbound)
	`relDist`(`q1`(open_lbs), `q3`(open_lbs))
	`std`(open_lbs) / `mean`(open_lbs)
	(`q3`(open_lbs) - `q1`(open_lbs)) / (`q3`(open_lbs) + `q1`(open_lbs))
Open nodes depths (4)	`mean`(open_ds) / SCIPgetMaxDepth
	`relDist`(`q1`(open_ds), `q3`(open_ds))
	`std`(open_ds) / `mean`(open_ds)
	(`q3`(open_ds) - `q1`(open_ds)) / (`q3`(open_ds) + `q1`(open_ds))