Zuse Institute Berlin ♦ Freie Universität Berlin

MASTER'S THESIS

Learning to Use Local Cuts

Author:

Matteo Francobaldi

Referees:

Dr. Timo Berthold
Prof. Dr. Ralf Borndörfer
Prof. Dr. Tim Conrad

A thesis submitted in fulfillment of the requirements
for the degree of Master of Science

in the

Department of Mathematics

Berlin, 8 October 2021

Contents

Acknowledgments

First of all, I would like to thank Dr. Timo Berthold, Prof. Dr. Ralf Borndörfer and Prof. Dr. Tim Conrad for supervising this thesis.
I wish to express my profound gratitude especially to you, Timo, for introducing me to this fascinating corner of mathematics, for teaching me so much of it, and for offering me your guidance. I feel lucky for having you as my mentor!
I would like to thank Dr. Gregor Hendel, for his collaboration and for his precious advices on my project.
I would like, moreover, to thank the staff of the Zuse Institute Berlin, for the friendly and welcoming atmosphere that they have guaranteed to me. It was a pleasure to be part of this institute!
Last, but certainly not least, I am grateful to my parents for their love and encouragement, and for supporting and trusting me in all my choices.

Abstract

Mixed-Integer Programming is a powerful framework for modeling and solving combinatorial optimization problems, arising from a wide range of real-world applications. For taking the non-trivial decisions that occur during the resolution of these hard problems, state-of-the-art solvers typically rely on a plethora of heuristics, often human-designed and tuned over time by using experience and data. Machine Learning, in this context, offers a promising approach to replace these handcrafted techniques with more principled and adaptive strategies. This is why, in recent years, the combinatorial optimization community has been witnessing a rapid growth of interest in the integration of data-driven routines into the existing optimization frameworks.
In this thesis, we propose a machine learning approach to address a specific algorithmic question that arises during the solving process of a mixed-integer linear programming problem, namely, whether to use cutting planes only at the root node or also at internal nodes of the branch-and-bound search tree, or equivalently, whether to run a cut-and-branch or rather a branch-and-cut algorithm. At least to the best of the author's knowledge, indeed, the MIP community is still suffering the lack of a satisfactory comprehension of this specific question for general MIP problems.
Within a supervised regression framework, we develop three machine learning models, *Linear Model*, *Random Forest* and *Neural Network*, for predicting the relative performance between the two methods, *local-cut* and *no-local-cut*. Hence, by conducting an extensive computational study over a large test bed of problems, we evaluate the produced strategies, and we show that they are able to provide, upon the existing policies, a significant improvement to the performance of the solver.

Introduction

Mixed-Integer Programming (MIP) is a cornerstone of Operations Research, Analytics and Computer Science, and undoubtedly represents one of the most advanced and powerful frameworks in tackling the hard problems that arise from Combinatorial Optimization (CO). Nowadays, it is used as a versatile decision-making tool in a large variety of application domains, ranging from renewable energies [KRBA16] to virtual reality [LNL16], from cancer detection [LZMW09] to food production [PAMAL15], from logistics [BS07] to project scheduling [BHL+10], among many others.

Machine Learning (ML), on the other side, consists in a powerful toolbox of techniques to turn massive amounts of data into valuable knowledge, conclusions and actions, and it has witnessed, during the last decade, such a tremendous development that it has now become a pervasive and ever-present technology in our society, employed in a broad range of applications that spans computer vision [GDDM14] and natural language processing [WSC+16], robotics [TMD+06] and cloud computing [Dem15], Internet of Things [MBB+20] and medicine [GM21].

Combined together, the prescriptive properties of Mixed-Integer Programming and the predictive capabilities of Machine Learning give rise to a powerful and forward-looking technology that, facilitated by the rapid growth of computing power and data availability, is conquering an ever more central role at the cutting edge of Artificial Intelligence.

In particular, one remarkable line of research that, in recent years, has been emerging from the cross-fertilization of these two fields consists in the investigation of the potential use of ML techniques for the improvement of CO – especially MIP – algorithms, with the former being seen as a promising approach for replacing those handcrafted heuristics, on which the latter heavily rely to take crucial decisions, with more principled and systematic strategies [LZ17, TAF20a]. This innovative and prolific area of research, commonly referred to as *Machine Learning for Combinatorial Optimization* [BLP21], represents the domain to which this thesis aims to belong and to contribute.

In this thesis, we present a machine learning approach to take a specific algorithmic decision while solving a mixed-integer linear program, namely, whether to use cutting planes only at the root node (*no-local-cut*), or also at internal nodes of the branch-

and-bound search tree (*local-cut*), or equivalently, whether to run a cut-and-branch or a branch-and-cut algorithm [PR91, Mit02, Pad05]. At least to the best of the author's knowledge, indeed, the scientific literature is still suffering a lack of publications addressing this specific question for general MIP problems.

Precisely, in a supervised regression framework, we represent each problem as a vector of features describing its combinatorial structure and mathematical formulation (*static features*), as well as its computational behavior (*dynamic features*), and we label it with a continuous variable specifying the relative performance between the two methods, local-cut and no-local-cut, while running on it. Hence, over a test bed obtained by joining several permuted copies of the *Benchmark Set* from MIPLIB 2017 [GHG+21], we develop different ML models, such as *Linear Model*, *Random Forest* and *Neural Network*, for predicting the speedup factor between the two approaches over the input MIP instance.

The ultimate goal of the present work, in particular, is to produce an effective predictive tool that can be strictly integrated into our reference MIP solver, namely, FICO Xpress [Xpr], in order to improve its performance. In fact, a variant of the random forest suggested in the present work has already been implemented by the development team of FICO Xpress; in particular, thanks to the promising results obtained after the preliminary testing phase conducted in this thesis, the intelligent agent will represent one of the main features to be released with the next version of Xpress [Com].

This thesis is structured as follows. We start by describing the fundamental notions and algorithms that constitute the backbone of Mixed-Integer (Linear) Programming, in the first chapter, and Machine Learning, in the second chapter, hence we introduce the two subjects at the background of our study. In the third chapter, we state and motivate the question that we want to answer, and we observe how natural an ML approach looks as a candidate to produce an effective and efficient strategy for our decision problem, given that such a strategy seems to be still missing from the state of the art. Hereto, we survey the major developments of the interplay between these two disciplines, before providing a theoretical illustration of the solution that we propose. In the fourth chapter, we formalize the ML methodology that we adopt to solve our algorithmic problem. Finally, in the fifth chapter, we present some of the most relevant computational experiments, conducted on our dataset to evaluate the quality of the produced models. We show, in particular, that the random forest is able to provide, to the average running time of the solver, a speedup of roughly 8%, which increases up to the encouraging value of 24% over those instances of our set that are particularly hard for the solver.

Chapter 1

Mathematical Optimization

In this chapter, we introduce the basic notions of Mathematical Optimization, representing the background of this thesis. In particular, we start from the definition of Mixed-Integer Programs, the type of optimization problems that constitute the object of our study. Then, we describe the two fundamental methods to solve such problems, namely, the Branch-and-Bound and the Cutting-Plane algorithm. Finally, we conclude the chapter by illustrating the general workflow of a modern MIP solver, with a special focus on FICO Xpress, the state-of-the-art software at the basis of our research.

1.1 Mixed-Integer Programs

A Mixed-Integer Program is defined as follows.

Definition 1.1. *A Mixed-Integer Program (MIP) is the problem of optimizing a linear function over a semi-discrete set of points, defined by a finite family of linear constraints. In canonical form, a MIP is expressed as follows:*

$$\begin{array}{ll} \textit{minimize} & c^T x \\ \textit{subject to} & Ax \leq b \\ & x \in \mathbb{Z}^{\mathcal{I}} \times \mathbb{Q}^{\mathcal{C}}, \end{array} \tag{1.1}$$

where $c \in \mathbb{Q}^n, b \in \mathbb{Q}^m, A \in \mathbb{Q}^{m \times n}$ and $\mathcal{I} \subseteq [n]$ is the subset of indices specifying the integer variables, while $\mathcal{C} = [n] \setminus \mathcal{I}$ indicates the continuous variables.

The function $c^T x$ is called the *objective* of the program, while the set $\mathcal{F}_{\text{MIP}} = \{x \in \mathbb{Z}^{\mathcal{I}} \times \mathbb{Q}^{\mathcal{C}} \mid Ax \leq b\}$ is the *feasible space*, and its points are called *feasible solutions*. A feasible solution x^* is *optimal* (or an *optimum*) if

$$c^T x^* = \min_{x \in \mathcal{F}_{\text{MIP}}} c^T x.$$

The restrictions forcing some of the variables to be integer are called *integrality constraints*. Without them, that is, when $\mathcal{I} = \emptyset$, the program (1.1) is commonly called

Linear Program (LP). On the other hand, if $\mathcal{I} = [n]$, then (1.1) is referred to as *Integer Program* (IP). Integer variables that are further restricted to the set $\{0, 1\}$ are called *binary variables*, and a program with only this kind of variables is known as *Binary Program* (BP). Note that, with a slight abuse of notation, in this thesis we will use the abbreviation MIP to refer to both "mixed-integer programming" and "mixed-integer program", and similarly with LP, IP and BP.

The integrality constraints are the ones responsible for the difficulty of solving MIPs, which are $\mathcal{NP}$-hard problems. As we have seen, a special case of mixed-integer programming is binary programming, and deciding the feasibility of a BP is one of the Karp's 21 $\mathcal{NP}$-complete problems [Kar72], from which it follows the $\mathcal{NP}$-hardness of MIPs. On the other hand, LPs are solvable in polynomial time by using the *ellipsoid method*, as first shown in 1979 by Khachiyan [Kha79].
This thesis focuses, in particular, on programs that involve integrality restrictions, even though LPs still have an important role, being them used as auxiliary problems by the solving algorithms. In fact, state-of-the-art MIP solvers usually attack a MIP with an LP-based approach, which consists in relaxing the integrality constraints and working with the so-called *Linear Relaxation* of the program, whose definition is provided below.

Definition 1.2. *Given a MIP as defined in (1.1), its Linear Relaxation is the LP obtained by dropping the integrality constraints from $\mathcal{F}_{MIP}$. Precisely,*

$$\begin{array}{ll} \textit{minimize} & c^T x \\ \textit{subject to} & Ax \leq b \\ & x \in \mathbb{Q}^n \end{array} \tag{1.2}$$

The feasible space of (1.2) is $\mathcal{F}_{\text{LP}} = \{x \in \mathbb{Q}^n \mid Ax \leq b\}$ and its points are called LP-feasible solutions. Such a solution $\hat{x}$ is an LP-optimum if

$$c^T \hat{x} = \min_{x \in \mathcal{F}_{\text{LP}}} c^T x.$$

The idea of working with the linear relaxation of the program is motivated by the following trivial observation: given a MIP as defined in (1.1), we have

$$\mathcal{F}_{\text{MIP}} = \mathcal{F}_{\text{LP}} \cap \mathbb{Z}^{\mathcal{I}} \subseteq \mathcal{F}_{\text{LP}},$$

which means that, for an LP-optimum $\hat{x} \in \mathcal{F}_{\text{LP}}$,

$$\text{if } \hat{x}_i \in \mathbb{Z} \text{ for all } i \in \mathcal{I}, \text{ then } c^T \hat{x} = \min_{x \in \mathcal{F}_{\text{MIP}}} c^T x.$$

In other words, if the optimal solution of the relaxation satisfies the integrality constraints, then the MIP is solved. This fact, although simple, represents a basic idea behind the two strategies at the heart of any modern MIP software, the *Branch-and-Bound* and the *Cutting-Plane* algorithm, which will be described in the following sections.

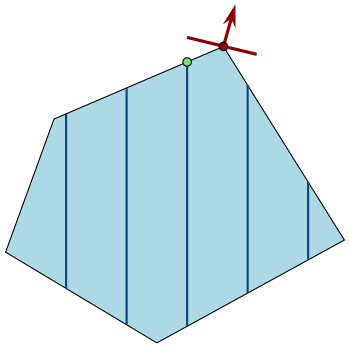

Figure 1.1: Geometric representation of a (5×2)-dimensional MIP. The union of the vertical segments is $\mathcal{F}_{\text{MIP}}$, the polyhedron represents $\mathcal{F}_{\text{LP}}$, the vector and the point in red are the direction of the objective and the LP-optimum $\hat{x}$, respectively, while the green point represents the optimum x^* of the MIP.

1.2 Branch-and-Bound

The Branch-and-Bound algorithm (B&B) is a general method to solve optimization problems, first proposed in 1960 by Alisa Land and Alison Doig [LD60]. In particular, its LP-based version, introduced by Dakin [Dak65] and described in this paper, is the fundamental procedure in the field of mixed-integer programming, representing the backbone of any state-of-the-art MIP solver.
Essentially, the algorithm relies on a *divide-and-conquer* approach: the given problem is recursively split into two or more subproblems of the same type, until they become easy enough to be solved directly; then, the best solution found among all the subproblems is returned as the global optimum. This procedure, which constitutes the *branching* part of the algorithm, is augmented with a *bounding* step: a subproblem is processed only if it satisfies a certain condition, dictated by two particular *bounds*, which are maintained during the whole process. The bounding operation allows to avoid a brute-force enumeration of all potential solutions of the problem, whose number is usually exponential and would lead to a very long running time.

The input of the algorithm is a MIP instance P, the output is either a solution of P, or the conclusion that P is infeasible, that is, its feasible space is empty. Beside these two possible outputs, a problem P can also be unbounded, meaning that the objective $c^T x$ can be arbitrarily decreased without violating any constraint. This case, however, is easily detected at the first relaxation, this is why we are assuming that P is bounded.

Algorithm 1: LP-based Branch-&-Bound

Input: an instance P as defined in (1.1). Assumption: P bounded
Output: one of the followings: 1. Infeasible: $x^* = none$, $u = \infty$
2. Optimal solution: x^*, u
Initialization: $\mathcal{L} \leftarrow \{P\}$, $x^* \leftarrow none$, $u \leftarrow \infty$

1. If $\mathcal{L} = \varnothing$ [*Termination*]
return x^*, u

2. Select $Q \in \mathcal{L}$, update $\mathcal{L} \leftarrow \mathcal{L} \setminus \{Q\}$ [*Node Selection*]

3. $Q_{rel} \leftarrow$ linear relaxation of Q [*Relaxation Solving*]
If Q_{rel} infeasible
goto *1*
$\hat{x} \leftarrow$ optimum of Q_{rel}, $l_Q \leftarrow c^T \hat{x}$

4. If $l_Q \geq u$ [*Bounding*]
goto *1*

5. $C \leftarrow \{j \in I \mid \hat{x}_j \notin \mathbb{Z}\}$ [*Integrality Checking*]
If $C = \varnothing$
Update $x^* \leftarrow \hat{x}$, $u \leftarrow l_Q$
goto *1*

6. Select $j \in C$ [*Variable Selection*]

7. $Q_1 \leftarrow Q \cup \{x_j \leq \lfloor \hat{x}_j \rfloor\}$, $Q_2 \leftarrow Q \cup \{x_j \geq \lceil \hat{x}_j \rceil\}$ [*Branching*]
Update $\mathcal{L} \leftarrow \mathcal{L} \cup \{Q_1, Q_2\}$
goto *1*

The algorithm maintains a list of open problems $\mathcal{L}$, together with a vector x^* and a value u, where it stores the best solution found so far, called *incumbent*, and the corresponding objective value, respectively.
The execution of the algorithm generates a *search tree* (Figure 1.3), where each *node* represents a subproblem or, in another fashion, a branch of the search space. The root node corresponds to the global problem P, while the leaves are either open subproblems, or ones already processed.

As long as the list of open problems is non-empty, the algorithm chooses a node Q and solves its linear relaxation Q_{rel}, to obtain an LP-optimum $\hat{x}$ with objective l_Q (if Q_{rel} is feasible).
Now, before keeping processing Q, the algorithm checks the *bounding condition* $l_Q \geq u$. This determines whether the node can be pruned, because the best objective value that we can achieve in Q is not better than the one provided by the incumbent, or if it deserves

to be further explored, since it might potentially supply an objective improvement.
Globally, the algorithm maintains a *primal bound*, given by the upper bound u, and a *dual bound*, which is the minimum of the local lower bounds l_Q of all the leaf nodes $Q \in \mathcal{L}$. The gap between these two bounds is a typical piece of information used to monitor the progression of the execution and to quantify the quality of the incumbent; in particular, it is called *primal-dual gap* and defined as follows.

Definition 1.3. *Let u and l be, respectively, the current primal and dual bound during a MIP solving process. Then, the primal-dual gap $\gamma \in [0,1]$ of u and l is*

$$\gamma(u,l) = \begin{cases} 0 & if \quad u = l = 0, \\ 1 & if \quad u \cdot l < 0 \ \vee \ u = none, \\ \frac{|u-l|}{\max\{|u|,|l|\}} & otherwise. \end{cases}$$

Thus, the search aims to close this gap, providing a proof of optimality.
If the bounding test is passed, the algorithm checks the feasibility of $\hat{x}$. If we are lucky and $\hat{x}$ turns out to be feasible, then we have found a new incumbent and we can move to another branch of the tree.
Otherwise, the node Q enters the last phase of the process: the algorithm chooses one of the fractional variables, say x_j, and splits Q into two subproblems Q_1 and Q_2, by adding once the constraint $x_j \leq \lfloor \hat{x}_j \rfloor$ and once $x_j \geq \lceil \hat{x}_j \rceil$. It is clear that the best between an optimum of Q_1 and an optimum of Q_2 corresponds to an optimum of Q, since the region $\{\lfloor \hat{x}_j \rfloor < x_j < \lceil \hat{x}_j \rceil\}$ does not contain any feasible solution, as shown in Figure 1.2.

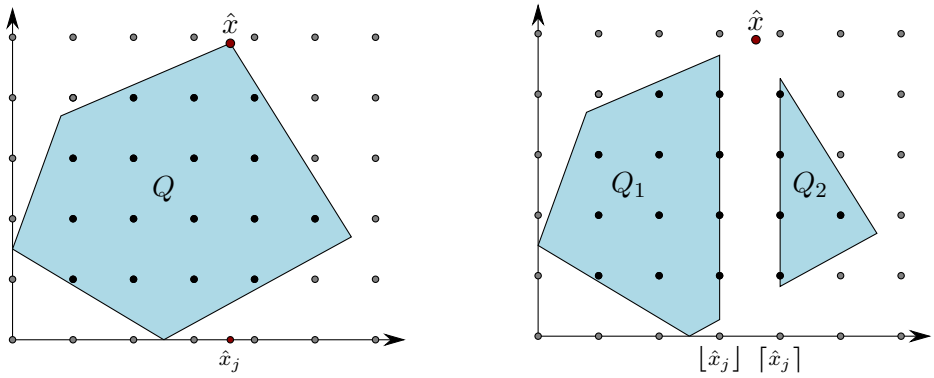

Figure 1.2: Branching on the fractional variable x_j

It is important to mention that the pseudocode above represents only a shell of the Branch-and-Bound algorithm, since it leaves a lot of room for freedom. In particular, two crucial decisions that are left open are the *Node Selection* and the *Variable Selection*. They have a huge impact on how quickly the bounds are updated, hence influencing the overall performance of the algorithm. Many efficient strategies, most of them for user-specified priorities, have been developed for taking these decisions

[SSR12, ABCC95, BGG+71, AKM05, AB09, BS13] (for an overview, we refer to the phd thesis of Tobias Achterberg [Ach07b]). In particular, they are among those topics in the field on which the research literature has recently seen a large involvement of Machine Learning [ZJLB20, EAB+20, LZ17].

Finally, we need to clarify that, in practice, the Branch-and-Bound algorithm is not used by the modern solvers as a standalone. In fact, the B&B skeleton is enriched with many subroutines, such as *heuristics* and *cutting planes*, used to improve the primal or the dual bound and to speed up the searching process. Cutting planes, in particular, play a special role in this thesis, and will be described in detail in the following section (for a comprehensive study of primal heuristics, we refer to the diploma thesis of Timo Berthold [Ber06]).

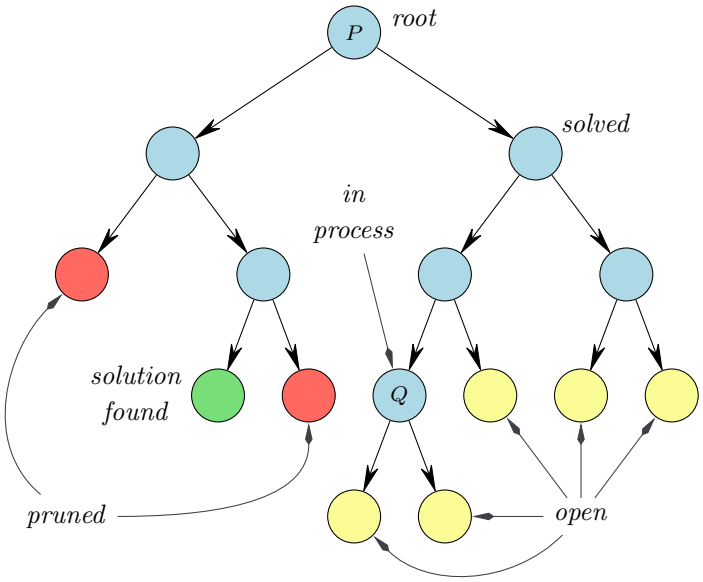

Figure 1.3: Search tree of a Branch-and-Bound execution.

1.3 Cutting Planes

Cutting-Plane methods represent a wide class of algorithms for solving mixed integer programs, whose use was proposed during the 1950s by Ralph Gomory [Gom58].
The basic idea behind these methods is to iteratively refine the linear relaxation of the program by adding linear constraints, until a satisfactory solution is found. These constraints are called *cutting planes* (shortly *cuts*), and the way in which they are constructed is what distinguishes these methods from each other.

The algorithm takes a MIP instance P as input and returns one of its optimal solutions

as output. As in the B&B, we are assuming that P is bounded, since the unboundedness would be easily detected at step *0*. Moreover, in describing the general Cutting-Plane algorithm, we also assume P feasible, to avoid infinite loops in case of *discrete infeasibility*, that is, the relaxation P_{rel} is feasible, but P is not.

Algorithm 2: Generic Cutting-Plane

Input: an instance P as defined in (1.1). Assumption: P bounded and feasible
Output: an optimum x^*
Initialization: $\mathcal{F} \leftarrow$ feasible space of P_{rel},
$\mathcal{F}_{\text{MIP}} \leftarrow$ feasible space of P

0. Solve P_{rel} and obtain LP-optimum x^*

1. If $x^* \in \mathcal{F}_{\text{MIP}}$
return x^*

2. Select inequality $\alpha^T x \leq \beta$ from set of candidates $\mathcal{C}$, so that
- ⋄ it is valid for $\mathcal{F}_{\text{MIP}}$
- ⋄ it is violated by x^*

3. Update $\mathcal{F} \leftarrow \mathcal{F} \cap \{x \mid \alpha^T x \leq \beta\}$

4. Solve new LP and update x^*, goto *1*

Thus, a cut is a linear inequality that fulfills two requirements: it is valid for the feasible space (i.e., satisfied by all feasible solutions of the problem), but violated by the current fractional optimum. Hence, the role of a cut is to "cut off" (or separate) the LP-solution, that we want to reject, from the feasible region of the MIP (see Figure 1.4).

The heart of the algorithm is the cutting phase, which consists of two operations: generating a set of valid cuts and selecting one (or more) from this set. As already mentioned, the cut generation is what makes the difference within this class of methods; there exist, in particular, many types of cuts (for an overview, see [Ach07b]), even though they can be grouped into two main categories: the one of matrix-based cuts ([Bal71], [CF96]), that are more generic, and the one of combinatorial cuts ([BZ78], [JP82]), which are, instead, more problem-specific. Once that, during the iteration, a set of candidates is available, the algorithm decides which ones should be used, that is, the ones to add to the formulation of the current problem. Taking this decision is not an easy task; moreover, it has a significant impact on the performance of the algorithm. This is why this problem, usually called *Cut Selection Problem*, has been, and continuous to be, intensively studied ([ACF07], [BCC96]). Moreover, like Node Selection and Variable Selection discussed before, also the Cut Selection Problem has been approached, in recent years, by means of Machine Learning techniques ([TAF20a]).

Theoretically, a cutting-plane method might be used as a standalone; for example, the Gomory's method was proved, by Gomory himself, to be able to solve any MIP, defined by rational data, in finite time, that is, an optimum can be found after adding

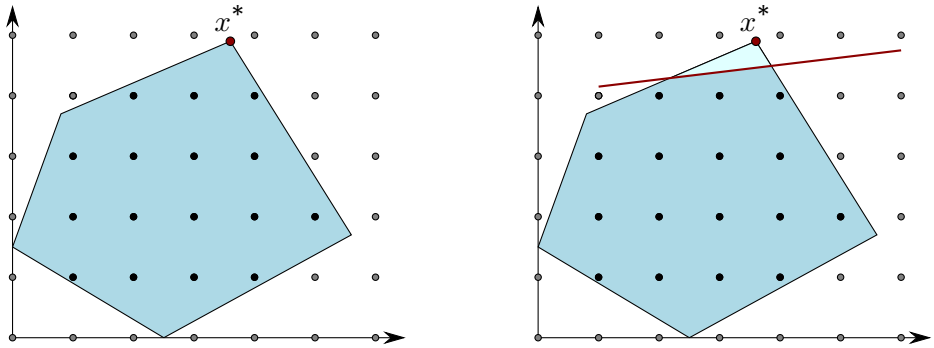

Figure 1.4: Adding a cut to "cut off" the fractional solution x^*.

a finite sequence of cuts [Gom58], [Gom60]. However, in practice, such methods alone may be incapable to solve difficult problems efficiently.

1.3.1 Use of Cutting Planes in B&B

Modern MIP solvers integrate cutting planes into the branch-and-bound main framework. The resulting hybrid algorithm, firstly formalized in 1991 by Padberg and Rinaldi [PR91], is referred to as *Branch-and-Cut* (B&C).

In this context, cutting planes are usually distinguished into *global cuts* and *local cuts*. We should say, however, that this classification is not uniformly defined through the MIP community and the research literature is still missing a standard convention in these regards.
In this thesis, we adopt the following terminology. The set of cutting planes, generated during a B&C execution, is partitioned into 3 groups:

- *global cuts*: they are generated at the root node, to separate the global LP-optimum from the global feasible space;
- *globally valid local cuts*: they are generated at an internal node, to separate the local LP-optimum from the local feasible space, but they are also valid globally, so they can be applied to all nodes;
- *locally valid local cuts*: they are generated at an internal node, to separate the local LP-optimum from the local feasible space, and they are only valid locally, that is, for that node and its descendants, hence they need to be deleted when the search leaves that branch of the tree.

Global cuts are, clearly, always valid for all subproblems and can be maintained during the whole search. Instead, the validity of a local cut, across the B&B tree, depends on whether the generation process takes the branching decisions into account. In particular,

there exist classes of cuts whose generation, by definition, never considers the branching decisions, so they are always valid globally, even if they are generated locally. On the other side, there are cuts whose generation can not disregard the variable bounds, hence, when they are generated as local cuts, they are intrinsically valid only locally. In other words, some cuts can only be classified as globally valid local cuts, some others only as locally valid local cuts. In the middle, there are classes of cuts that can be locally generated either as globally valid or as locally valid. In this case, the approach adopted depends exclusively on the solver or the user. A survey on the different types of cuts is proposed in [MMWW02], while a study on their validity within the B&B framework is illustrated in [Pad05].
We note that, sometimes, it is possible to manipulate cutting planes in order to extend their validity: there are classes of cuts that, even if generated as locally valid during the search, can be made globally valid. Similarly, it is also possible for some types of cuts, only valid for a particular branch of the tree, to be transferred to other branches. In other words, their validity is not extended to the global problem, but simply tailored to other nodes (an example on the use of this technique can be found in [WB21]). The choice of whether to adopt such techniques to manipulate and transfer cuts from nodes to nodes depends, again, on the solver or the user.
We clarify that, for the scope of this thesis, the difference between the two types of local cuts is not relevant. Hence, to simplify the terminology, for the remainder of this paper we will refer to any locally generated cut simply as local cut, regardless of its validity, unless explicitly mentioned otherwise.

The cutting strategy, within the B&C scheme, is determined by several algorithmic decisions, that should be taken carefully according to the problem structure. Important issues concern the classes of cuts to use and the aggressiveness of the cutting activity: which types of cuts should be generated? Should cuts be added at all nodes or only at some of them? Should the cutting procedure be limited to a certain depth of the search tree? How many cuts should be added in each cutting phase? This thesis, in particular, addresses one specific question from this list, which will be extensively discussed in the following chapters.
As for any other routine involved in the solving process of a MIP, designing an efficient cutting strategy is a matter of trade-offs: on one side, cutting planes can be very effective in improving the dual bound, on the other side, their generation does not come for free and, moreover, their inclusion enlarges the size of the matrix, hence increasing the computational effort necessary to solve the linear relaxations. An important consideration is that, in general, cutting planes are known to be particularly powerful at the beginning of the search, while they tend to lose their efficiency when the depth of the tree increases.
In general, it is not a good idea to generate all possible types of cuts, since usually only some of them are really suitable for the problem to solve, while the others might rather be time-wasting. For this reason, a hierarchy of generation routines is often adopted [JRT95].
Moreover, cutting planes are not added at all nodes of the tree but only at some of them,

depending on the local state or on a certain fixed frequency. The cutting activity is normally performed more aggressively at superficial levels of the tree, while it is progressively relieved at deeper levels. For example, cuts might be applied to every 2^k-th level (default of XPRESS).
In addition, a maximum depth for the cutting procedure is usually fixed, beyond which cutting planes are no longer applied. A common strategy consists in using cuts only at the root node; this version of the algorithm is usually called *Cut-and-Branch* (C&B), and sometimes represents the preferred alternative [Mit02].
Moreover, a criterion is adopted to limit the number of cuts during a cutting phase, both for controlling the size of the matrix and for avoiding the so-called *tailing-off* [PR91], which occurs when, at some point during the loop, cutting planes no longer provide any significant improvement to the dual bound of the current node.
Finally, another technique, used to maintain the size of the matrix reasonable, consists in removing, from the formulation of the problem, cutting planes that are no longer effective.
For a more detailed description of the use of cutting planes and the implementation of the Branch-and-Cut algorithm, we refer to [CMLW97, Mar01, FM05, Ach07b].

The above discussion on the cutting procedure and, especially, the distinction between global and local cuts, play a crucial role in this thesis; in particular, the benefits provided by the use of local cuts will be discussed and investigated in the following chapters.

1.4 Practical MIP Solving & FICO Xpress

MIP solving, in practice, consists in a complex combination of sophisticated techniques. Besides the two algorithmic principles described in the previous sections, Branch-and-Bound and Cutting-Plane, many other ingredients contribute to the solving process, each of them playing a crucial role for the effectiveness of the solver. The general workflow, adopted by all the state-of-the-art MIP solvers, is illustrated in Figure 1.5.

First of all, the input problem is manipulated by several algorithms that identify and remove redundant rows and columns, in order to reduce the size of the matrix and simplify the formulation of the problem. The collection of these techniques is usually referred to as *presolve*.
The initial presolving step is usually followed by a heavy cutting stage, during which the feasible region of the linear relaxation is tightened as much as possible before starting the main B&B loop. However, as explained in Section 1.3.1, cutting planes might also be applied deeper in the search tree.
Other important components of the solving scheme are *primal heuristics*, namely, methods to find a solution in a reasonable amount of time, but without any warranty of optimality (see [Ber06]). Primal heuristics find place almost everywhere within the solving process: they might be alternated with cuts during the cutting phase, or executed between the end of the current node and the selection of a new one. Their role is to produce good feasible solutions and, consequently, to improve the primal bound.

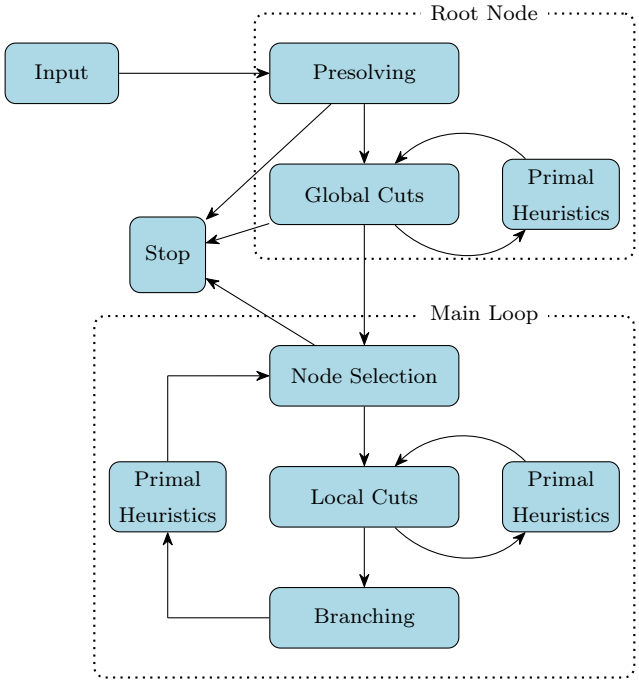

Figure 1.5: General workflow of a MIP solver.

We should clarify that the flowchart in Figure 1.5 is not exhaustive about the solving process of a MIP; there are, indeed, other techniques that are usually employed within a modern solver, such as *domain propagation* and *conflict analysis*, whose description, however, goes beyond the scope of this thesis. Moreover, we should also say that, usually, MIP solvers provide access to the single components of the solving process, allowing advanced users to manipulate them and, eventually, deactivate them completely.
An extensive description of all the components of a MIP solver can be found in [Ach07b]; inspections of the evolution of MIP computation over the years are provided both in [Lod10] and [Bix12].

The intense usage of mixed-integer programming in a wide range of domains has motivated the need of developing efficient MIP solvers and raised the commercial interest around the field. Today, the best software poducts for solving MIPs are developed for commercial purposes, such as Cplex [Cpl], Gurobi [Gur], Mosek [Mos] and Xpress [Xpr]. Valid solvers, however, are also available for academic, noncommercial uses, like Cbc [Cbc], lpsolve [lps], Glpk [GLP], Symphony [Sym] and SCIP [SCI]. A list of currently available tools for linear and mixed-integer programming, together with a description of their features, is published every two years by Robert Fourer. The last survey, at the time of writing this thesis, dates back to 2019 [Fou19].

The technology at the basis of this thesis is FICO Xpress, which represents, for us,

both the framework of our computational study and the target of our research, whose final goal, indeed, is to improve the overall performance of the solver itself.
XPRESS is a fast, robust and scalable mathematical optimization software, representing the core component of the Xpress Optimization Suite, developed and distributed by FICO. Originally released in 1983 exclusively as an LP solver, XPRESS is currently able to solve a broad range of optimization problems, with mixed-integer programs being, of course, among those. The solver can be interfaced via the command line, via a graphical interface, as well as through a library that is callable from the major programming platforms. XPRESS offers flexible access to the internal data structures and great control over the execution of the algorithms, allowing the user to manipulate the problem and customize the solving process. As we will see, particularly relevant, for this thesis, is the use of cutting planes with XPRESS, where the cutting procedure is implemented as follows.
While generating cut, the solver always tries to use all local information from the current node, including the variable bounds. This means that, when possible, it prefers to generate and add what we defined as locally valid local cuts (see Section 1.3.1). Moreover, in XPRESS, there is no attempt of manipulating the validity of cuts in order to adapt them to other nodes: local cuts are simply removed from the matrix of the problem when they are no longer valid. The solver provides a suite of controls to manage the actual cutting scheme: *cutstrategy* to control the number of cuts to add in each cutting phase, *cutfreq* to determine the levels of the tree where cuts should be generated, *cutdepth* to set the maximum depth for the cutting activity, and some other parameters to separately control the most-known classes of cuts. Finally, for a more advanced customization of the use of cutting planes, different routines are available in the *Cut Manager* framework to directly add cuts at specific nodes, to remove them, or to manipulate the *cut pool*, the data structure used to store cutting planes during the search.
For a detailed description of FICO Xpress, we refer to the official documentation [Xpr].

Chapter 2

Machine Learning

Machine Learning is the study of algorithms that use experience, in the form of data, to attain and improve the ability to perform certain tasks automatically, that is, without being explicitly programmed. Depending on the learning paradigm and the form of data used, machine learning is traditionally divided into three broad categories: *Supervised Learning*, *Unsupervised Learning* and *Reinforcement Learning*.
In the first section of this chapter, we formalize the general supervised learning paradigm, which represents the approach adopted in this thesis. In the remaining sections, instead, we illustrate the three specific learning algorithms used in our research, namely, *Linear Model*, *Random Forest* and *Neural Network*.

2.1 Supervised Learning

Supervised learning is the problem of inferring the relation between two variables, by experiencing a sample set of observations that partially describes this relation.
Formally, the supervised learning framework consists of the following components.

- An input space $\mathcal{X}$, whose elements are called *instances*. Each instance in $\mathcal{X}$ is represented by a vector $x = (x_1, \dots, x_n)$, whose entries are said *features*.
- An output space $\mathcal{Y}$, whose elements are called *labels*. The space $\mathcal{Y}$ can be either a finite set of *classes* or a continuous set; in the former case the problem is called *classification*, in the latter it is referred to as *regression*.
- An unknown relationship between the two spaces, that we model via a *joint probability distribution* $\mathcal{P} = \mathcal{P}(x, y)$ over the space $\mathcal{X} \times \mathcal{Y}$. Precisely, $\mathcal{P}$ can be seen as composed of two parts: a *marginal distribution* $\mathcal{P}_x$ over $\mathcal{X}$, which determines how likely it is to encounter any input x, and a *conditional distribution* $\mathcal{P}((x, y)|x)$ over $\mathcal{Y}$, which specifies the probability that the input x is labeled with the output y.
- A *dataset* $\mathcal{D} = \left\{\left(x^{(i)}, y^{(i)}\right)\right\}_{i=1}^{i=N} \subseteq \mathcal{X} \times \mathcal{Y}$, providing an empirical description of the

input-output relation. The pairs in $\mathcal{D}$ are sampled from the distribution $\mathcal{P}$, that is, $\left(x^{(i)}, y^{(i)}\right) \sim \mathcal{P}$ for all $i \in [N]$.

- A *model space* $\mathcal{M} = \{M : \mathcal{X} \longrightarrow \mathcal{Y}\}$, defining the form of the model that we want to use in order to estimate the relation between $\mathcal{X}$ and $\mathcal{Y}$. This space, especially in statistical learning literature, is also known as *hypothesis space*, while its elements are also referred to as *hypothesis*.
- A *loss function* $l : \mathcal{Y} \times \mathcal{Y} \longrightarrow \mathbb{R}_+$, used as a measure of success for the learning algorithm, that is, to quantify the ability of the model to correctly estimate the provided instance: for $(x, y) \sim \mathcal{P}$, $l(M(x), y)$ measures how "far" the estimation $M(x)$ is from the correct label y.

The goal of a supervised learning algorithm is to approximate the relation between $\mathcal{X}$ and $\mathcal{Y}$, that is, to produce a model M such that $M(x) \approx y$ for each pair $(x, y) \sim \mathcal{P}$. Precisely, this goal is formalized as the following minimization problem:

$$\min_{M \in \mathcal{M}} L_{\mathcal{P}}(M) = \min_{M \in \mathcal{M}} \mathbb{E}_{(x,y)\sim\mathcal{P}}[l(M(x), y)], \tag{2.1}$$

where the objective $L_{\mathcal{P}}(M)$ is called *generalization error*.
This goal, however, is not achievable directly, since the distribution $\mathcal{P}$ is unknown and inaccessible. Therefore, it is attained empirically by using the available information $\mathcal{D}$, that is, by solving the following problem:

$$\min_{M \in \mathcal{M}} L_{\mathcal{D}}(M) = \min_{M \in \mathcal{M}} \frac{1}{N} \sum_{i=1}^{i=N} l\left(M(x^{(i)}), y^{(i)}\right). \tag{2.2}$$

where $L_{\mathcal{D}}(M)$ is said *empirical error*, while the process of solving (2.2) is called *learning*.

Now, on one hand, this formulation of the learning paradigm emphasizes the large interplay between the two scientific fields at the background of this thesis, machine learning and mathematical optimization. The former, in particular, heavily relies on the latter, given that learning, essentially, means solving an optimization problem.
On the other hand, however, the scheme described above also highlights the element that separates machine learning from pure optimization, namely, the desire for *generalization*. Indeed, rather than performing perfectly on the observed data, the central challenge of machine learning is to perform well on new, previously unseen data. In machine learning, we minimize the empirical error with the final goal of reducing, as much as possible, the generalization error. In another fashion, we try to learn, through the observed data, to make predictions on future data; for this reason, we also refer to a learning model M as a *predictor*, and to the estimated value $M(x)$ as the *prediction* of M for $x \in \mathcal{X}$.

For a more extensive discussion on the general learning theory, we refer the reader to the popular textbook "Understanding Machine Learning: From Theory to Algorithms" [SSBD14], which proposes detailed descriptions of the basic notions of statistical learning, such as *Empirical Risk Minimization* and *PAC Learning*.

2.1.1 Train/Test Set

In practical development of a machine learning system, the dataset $\mathcal{D}$ is split into two disjoint subsets: one of them is used for *training* the model, the other for *testing* its generalization abilities; accordingly, the two subsets are called *train set* and *test set*, and denoted as $\mathcal{D}_{train}$ and $\mathcal{D}_{test}$, respectively. Moreover, the error of the model on $\mathcal{D}_{train}$ is called *train error*, the one on $\mathcal{D}_{test}$ is said *test error*; the former represents the empirical error, the latter is used to approximate the generalization error. Note that, in the development of learning algorithms, the way in which the dataset is split represents a crucial decision, which should be taken carefully according to the nature of the dataset.

2.1.2 Underfitting vs Overfitting

The problem to minimize the train error and, in the same time, to also guarantee a low test error, leads to two fundamental notions of machine learning, *underfitting* and *overfitting*, hence to one of the major issues faced by any learning procedure, the need of finding the *trade-off* between the two.
We say that the model is *overfitting* the data when it performs well on the train set, but shows an unsatisfactory performance on the test set, that is, the gap between the train and the test error is too large. Instead, if both errors are high, then we say that the model is *underfitting*, that is, it is not even able to fit the training data accurately.
The risk to underfit or overfit depends on the *capacity* of the model adopted, where the capacity of a model is, informally, its ability to fit a wide variety of datasets or, more briefly, its learning power. When the capacity is too low with respect to the problem to solve, the model might not be able to learn the recurrent patterns in the data, hence resulting in underfitting. On the other hand, when the capacity is too high, the model might risk to capture noise and fluctuations in the data that deviate it from the underlying structure, hence resulting in overfitting. Thus, the challenge consists in choosing the appropriate capacity for the true complexity of the data, that is, in finding the *trade-off* between under- and overfitting.

For a more formal discussion on the notion of capacity, we refer again to the field of statistical learning [SSBD14], which provides several tools to effectively quantify the learning capabilities of a given model, such as the well-known Vapnik-Chervonenkis dimension (VC dimension), a technique particularly suitable for binary classifiers [VC71].

2.1.3 Hyperparameters & K-Fold Cross-Validation

The behavior of a learning algorithm is usually controlled by a set of values called *hyperparameters*, used to specify the architecture of the adopted model and to govern the behavior of the training procedure. Just to give an example, a typical hyperparameter of a deep learning algorithm is the number of hidden layers of the neural network, as well as the number of hidden units in each hidden layer (see Section 2.4).
Hyperparameters can affect the speed of the training algorithm and, even more impor-

tantly, the accuracy of the final model. For this reason, in the development of any learning algorithm, choosing an appropriate hyperparameter configuration represents a crucial decision, and the problem of taking such a decision is, as the training itself, an optimization problem, usually referred to as *hyperparameter tuning* (or *model selection*). Clearly, the hyperparameters should be tuned before starting the actual training, that is: a model is trained and evaluated for each hyperparameter configuration, hence, the configuration providing the highest performance is selected and used for training a final model. In the model selection phase, the evaluation of each model can not be done on the test set $\mathcal{D}_{test}$, since this would bias the final predictor. This is why the hyperparameters of the learning algorithm are commonly tuned over $\mathcal{D}_{train}$ by using *k-fold cross-validation*.

K-fold cross-validation is a general technique for training and evaluating a machine learning model, commonly used when the ground dataset is not plentiful enough to be split into two parts, hence allowing the execution of the training/evaluation process over two disjoint sets. The procedure works as follows. The given dataset is partitioned into k subsets (*folds*) of the same size (for simplicity, assume the size of the set is divisible by k); for each fold, a model is trained on the union of the other folds and tested on this fold. Thus, the final result of the cross-validation is obtained by averaging the results of all validations.
In model selection, each hyperparameter setting is evaluated by using cross-validation over the train set $\mathcal{D}_{train}$, hence the setting minimizing the error is returned as the optimal one. A pseudocode of this process is provided in Algorithm 3. The procedure takes as inputs a set $\mathcal{D}_{train}$, a space of possible hyperparameters Θ, an integer k and a learning algorithm A. In particular, for a training set and a tuple of hyperparameters, A returns a model trained on that set and with those hyperparameters.

Algorithm 3: Model Selection with K-Fold Cross-Validation

Input: integer k
hyperparameter space Θ
learning algorithm A
train set $\mathcal{D}_{train} = \left\{(x^{(1)}, y^{(1)}), \ldots, (x^{(t)}, y^{(t)})\right\}$ with $t/k \in \mathbb{Z}$

Output: optimal hyperparameter configuration $\theta^* \in \Theta$

1. Split $\mathcal{D}_{train}$ into k folds of same size t/k: $F_1, \ldots, F_k$

2. For $\theta \in \Theta$:

For $i \leftarrow 1$ to k:

$M_\theta^i \leftarrow A\left(\mathcal{D}_{train} \setminus F_i; \theta\right)$

$L_{F_i}\left(M_\theta^i\right) \leftarrow \frac{1}{|F_i|} \sum_{(x,y)\in F_i} l\left(M(x), y\right)$

$Err(\theta) \leftarrow \frac{1}{k} \sum_{i=1}^{i=k} L_{F_i}\left(M_\theta^i\right)$

3. Return θ^* such that $Err(\theta^*) = \min_{\theta\in\Theta} Err(\theta)$

After that the hyperparameters have been tuned, the configuration selected as optimal is used to train a final model, this time over the whole train set $\mathcal{D}_{train}$. We clarify that not all learning algorithms have hyperparameters; obviously, in this case, there is no need for model selection. For a more detailed description of model selection and cross-validation we refer to [SSBD14].

In the following sections of this chapter, we will describe the three machine learning models used in this thesis, namely, *Linear Model*, *Random Forest* and *Neural Network*. In particular, we will treat regression and binary classification: in the former we will always have $\mathcal{Y} = \mathbb{R}$, in the latter $\mathcal{Y} = \{-1, 1\}$; the input space, instead, will always be $\mathcal{X} = \mathbb{R}^n$.

2.2 Linear Model

A linear model is a function that involves a linear relationship between the input variables and the output labels. Linear models represent an old class of predictive functions, developed in the pre-computer age of statistics. They are, however, still widely used in machine learning, being them intuitive, simple to train and easy to interpret.
Precisely, a linear model $LM_{w,b}$, between the input space $\mathcal{X} = \mathbb{R}^n$ and the output space $\mathcal{Y}$, is defined as

$$LM_{w,b}(x) = \phi\left(w^T x + b\right) = \phi\left(\sum_{i=1}^{n} w_i x_i + b\right),$$

where $w \in \mathbb{R}^n$ and $b \in \mathbb{R}$ are the parameters of the model, while ϕ depends on the nature of the output space: when $\mathcal{Y} = \mathbb{R}$ (regression), ϕ is simply the identity function, when $\mathcal{Y} = \{-1, 1\}$ (binary classification), ϕ is a function that transforms continuous values into binary, such as the sign function. In particular, a space of linear models is determined by specifying ϕ.
Geometrically, the model $M_{w,b}$ can be thought as the hyperplane

$$w_1 x_1 + \cdots + w_n x_n + b = 0,$$

whose role depends on the nature of the learning task: while, in regression, the hyperplane is used to directly fit the data, in binary classification, it is meant as a separator of the two classes (see Figure 2.1).

Being a parametric function, a linear model is trained by estimating its parameters from the data, that is, by searching for a parameter configuration that minimizes the train error. Following the scheme proposed in Section 2.1, this involves the following steps: define a model space (by choosing a suitable ϕ), decide a proper loss function to compute the error and design a procedure to solve the corresponding optimization problem. A large number of machine learning algorithms have been developed to train linear models. In regression, a typical learning technique is *Least Squares* [Leg05], while in classification, common choices are *Perceptron* [Ros58], *Support Vector Machine* [BGV92] and *Logistic Regression* [M.D44, M.D51]. In the latter, in particular, the learning model is

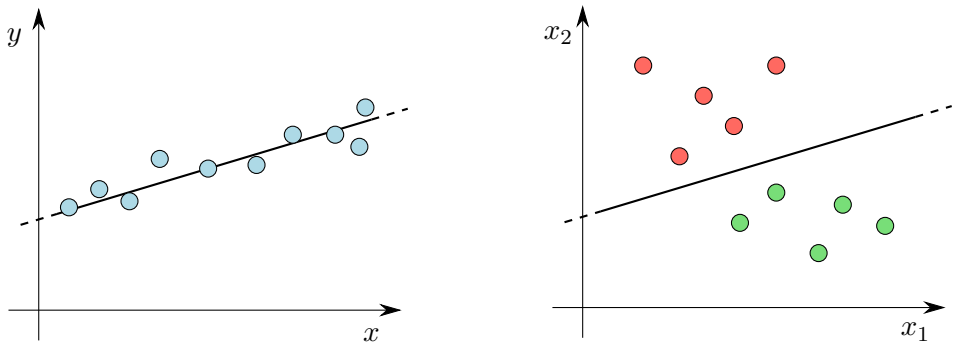

Figure 2.1: On the left, a linear regressor fitting the data; on the right, a linear classifier separating the two classes. Note: in the regression example, each input instance is defined by a single feature x and labeled by the continuous label y, in the classification case, instead, the instances are specified by two features x_1 and x_2, and classified as green or red.

a function from the input space $\mathbb{R}^n$ to the interval $[0, 1]$, used to predict the probability that the input instance belongs to one of two possible classes. In other words, the logistic regression algorithm learns a regression model, which is used, however, as a binary classifier.

2.3 Random Forest

Firstly introduced in 1995 by Tin Kam Ho [Ho95], *random forests* are predictive models commonly used in machine learning, suitable for both regression and classification problems. They represent a special case of *ensemble methods*, that is, methods consisting in gathering different predictors together, in order to give rise to a more powerful one. Precisely, in the case of random forests, the single predictors in the ensemble are *decision trees*.

2.3.1 Decision Tree

A decision tree is a popular support tool for decision analysis; it consists of a set of decision rules (if-else statements) and can be represented in the form of a tree-like structure. The first learning algorithm for decision trees was proposed for regression in 1963 by Morgan and Sonquist [MS63], then extended to classification in 1972 by Messenger and Mandell [MM72]. Precisely, the non-leaf nodes of the tree correspond to conditional tests over specific features from the input space $\mathcal{X}$, such as $x_j \leq \alpha$, where α is a certain threshold, the branches represent the outcomes of these tests, while the leaves correspond to target values from the output space $\mathcal{Y}$. The labeling process, for a given input x, consists in traversing the graph from the root node to a leaf node, following a path that is dynamically determined by the tests performed over the features of x, that is, each node

in the succession depends on the decision taken in the preceding node. Hence, the input is labeled with the value stored in the leaf where the traversal ends up.
More formally, a tree model $T\colon \mathcal{X} \to \mathcal{Y}$ is a piecewise constant function that partitions the feature space $\mathcal{X}$ into axis-aligned rectangular regions, each one associated with a certain value in $\mathcal{Y}$. With such a geometrical interpretation, the assignment procedure consists in identifying the region of the partition where the input x falls, hence in returning the corresponding value as the output $T(x)$. Precisely, each edge of the tree corresponds to a region of $\mathcal{X}$, while the node receiving the edge represents a split of this region into two subregions. An example of the assignment process of a decision tree, for a given input instance, is illustrated in Figure 2.2.

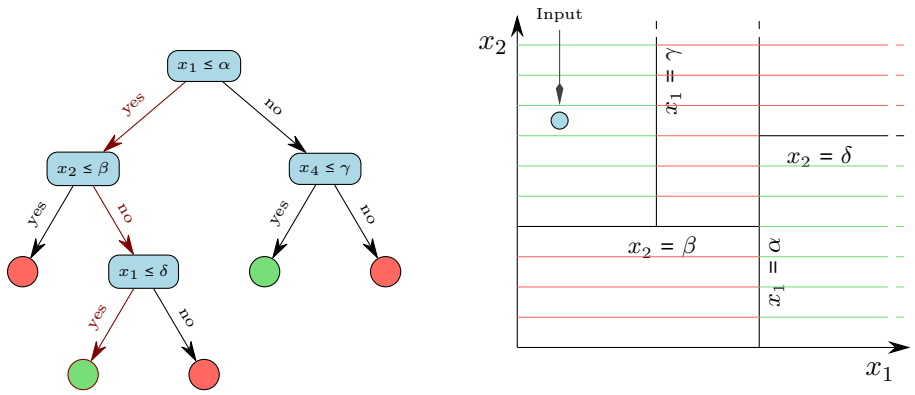

Figure 2.2: On the left, a decision tree for a binary classification problem with two features, x_1 and x_2, and two classes, red and green; on the right, the corresponding partition of the 2-dimensional feature space. Traversing the tree to classify the input point is equivalent to locating the region which the point belongs to.

Now, as for other machine learning models, the process of "growing" a tree, over a given train set, can be formulated as an optimization problem, which consists in searching for the tree model that minimizes the train error, defined according to a certain loss function. It turns out, however, that solving such an optimization problem is computationally hard in several variants (see for example [HR76, HJLT96]). For this reason, practical algorithms for training decision trees rather rely on heuristic techniques, such as the greedy approach, where the tree is gradually grown by taking, at each node, locally optimal decisions. Clearly, such techniques are not able to guarantee the optimal tree model for the learning problem, however, they have proved to work quite well in practice.
Many different algorithms have been developed for efficiently growing decision trees, both for regression and classification problems, such as *ID3* [Qui86] and its extension *C4.5* from Quinlan [Qui93], *CART* from Breiman et al. [LBS84] and *CHAID* from Kass [Kas80]. For a survey on the available techniques in decision tree learning, as well as for

an overview of the historical developments in the field, we refer to [Loh14].

Decision trees are appreciated for their interpretability and flexibility: they are intuitive, graphically visualizable and easy to explain, moreover, they can handle both numerical and categorical data, hence they are suitable for both regression and classification problems. On the other hand, however, they might suffer for non-robustness and over-complexity: small modifications in the data might be reflected into large changes in the tree, moreover, while growing, they might evolve into too elastic structures, which are obviously prone to overfitting.
The generalization error can be reduced through the use of different mechanisms, such as by adopting stopping criterions to prevent an excessive growth of the tree, or by pruning it after its construction. In particular, one of the most efficient approaches to correct the overfitting problem is a specific type of ensemble method, namely, *random forest*

2.3.2 Random Forest

A random forest is an ensemble of decision trees, whose outcomes are derived independently of each other and then combined together into a single prediction. Precisely, the output of the ensemble, for a given input, is obtained by averaging the predictions from the individual trees (in regression), or by taking the majority vote (in classification).
The trees in the forest are grown by using a combination of two learning techniques: *bagging*, short for *bootstrap aggregation*, and *random feature selection*. For a description of these procedures we refer to [Ho95, Bre96, Bre01], while here we simply mention the benefits provided by their use: the former allows to reduce the overall sensitivity of the forest to noise in the training data, while the latter prevents the trees in the ensemble from being highly correlated. Combined together, the two techniques correct the typical habit of a single decision tree for overfitting.

To control the learning process of a random forest, the most important hyperparameters are the number of trees in the bagging procedure, the number of features which are randomly selected, and the maximum depth for the growth of each tree. Clearly, their configuration strongly depends on the problem to solve and the size of the train set. As for other learning algorithms, the optimal setting is usually found by means of cross-validation.

2.4 Neural Network

An *artificial neural network*, or briefly *neural network*, is a model of computation loosely inspired by the human brain. The study of these models, and in particular their use in solving learning problems, delineates an entire subfield of machine learning, called *Deep Learning*. The starting point of this research area dates back to 1943, when McCulloch and Pitts published the first mathematical formulation of a biological neuron [MP43]. The first learning algorithm for this model, called *Perceptron*, was instead proposed in

1958 by Rosenblatt [Ros58].
Different types of neural networks have been developed for different applications; in particular, the most common type, which is also the one used in this thesis, is the *feedforward neural network*.

A feedforward neural network is defined as a tuple (G, σ, w, b), where $G = (V, E)$ is a directed acyclic graph (*architecture*), $\sigma\colon \mathbb{R} \to \mathbb{R}$ is a nonlinear function (*activation function*), $w\colon E \to \mathbb{R}$ is a function over the edges (*weight function*), and $b\colon V \to \mathbb{R}$ is a function over the nodes (*bias function*). The graph, whose nodes are called *neurons*, is structured as a sequence of *layers* $L_1, \ldots, L_d$, defining a partition of the node set, $V = \cup_{l=1}^{d} L_l$. In particular, L_1 is called the *input layer* and contains as many neurons as the dimension of the input space $\mathcal{X}$, L_d is said the *output layer* and its size matches the dimension of the output space $\mathcal{Y}$, the internal layers are referred to as *hidden layers* and consist of an arbitrary number of neurons; finally, the integer d is called the *depth* of the network. Each edge $(u, v) \in E$ is a connection between two neurons in two consecutive layers, that is, $u \in L_{l-1}$ and $v \in L_l$, for some $l = 2 \ldots, d$. We assume that the network is *fully-connected*, that is, each subgraph of the network, induced by any pair of consecutive layers, is a complete bipartite graph. Note that, if desired, we can always disconnect two neurons by fixing the weight on the corresponding edge to 0. An example of the architecture of a feedforward neural network is provided in Figure 2.3.

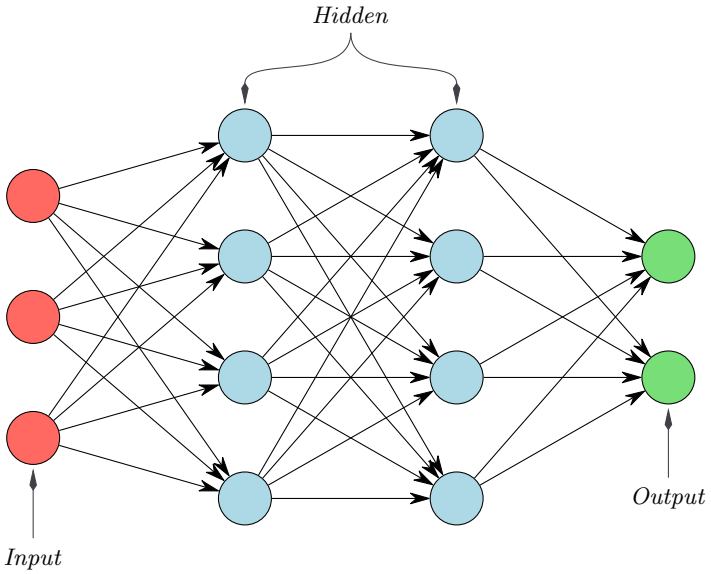

Figure 2.3: A feedforward neural network of depth 4, with 1 input layer of 3 neurons, 1 output layer of 2 neurons, and 2 hidden layers of 4 neurons each.

For a given instance $x \in \mathcal{X}$, the assignment process works as follows. The instance

is fed into the network from the initial layer, where the computational flow starts to propagate in the forward direction. In particular, each layer receives a vector from the previous layer, applies a certain transformation to it, and sends it to the next layer. Hence, the process terminates when the flow reaches the last layer, which returns the final vector as output. The fact that the information flows without feedbacks is reflected in the property of the graph to be acyclic, and explains why these networks are referred to as feedforward. To precisely define the computations executed during the process, we adopt the following terminology:

- n_l is the size of the l-th layer of the network, for $l \in [d]$;
- v_i^l is the i-th neuron of the l-th layer, b_i^l is the bias associated to v_i^l, for $l \in [d]$ and $i \in [n_l]$;
- $w_{i,j}^l$ is the weigth of the edge directed from v_j^l to v_i^{l+1}, for $j \in [n_l]$, $i \in [n_{l+1}]$ and $l = 1, \dots, d-1$.

Now, during the computational flow, each neuron v_i^{l+1} receives a vector of values x^l, sent by the previous layer, and a vector of weights w_i^l, carried out by the incoming edges, and combines the two vectors into the following output:

$$x_i^{l+1} = \sigma\left(\sum_{j=1}^{n_l} w_{i,j}^l x_j^l + b_i^{l+1}\right).$$

Hence, the n_{l+1} outputs, produced by the neurons of the $(l+1)$-th layer, are sent to the next layer as components of the vector x^{l+1}. In other words, a neuron is a vector-to-scalar function and constitutes the basic computational unit of a neural network. It is, in particular, a linear function, as described in Section 2.2. An example of artificial neuron is illustrated in Figure 2.4. Overall, the process starts with the vector x^0 received by the first layer, corresponding to the input instance from the space $\mathcal{X}$, and terminates with the vector x^d returned by the last layer, representing the output label from $\mathcal{Y}$ assigned to x. Note that the input layer has no effects on its input, i.e., $L(x^0) = x^1$.

By combining together the action of all neurons in the same layer, we can interpret each layer L_{l+1} as a vecto-to-vector function $L_{l+1}\colon \mathbb{R}^{n_l} \to \mathbb{R}^{n_{l+1}}$, defined as follows

$$x^{l+1} = L_{l+1}(x^l) = \sigma(W^l x^l + b^{l+1}),$$

where x^l is the output of the l-th layer, $b^{l+1} = (b_i^{l+1})_{i \in [n_{l+1}]}$ is the bias vector of the $(l+1)$-th layer, $W^l = (w_{i,j}^l)_{(i,j) \in [n_{l+1}] \times [n_l]}$ is the weight matrix between the two layers, and σ is meant here has a vector function that works element-wise. Hence, with the matrix notation adopted, we can write the neural network as a single function between the input space $\mathcal{X}$ and the output space $\mathcal{Y}$, obtained by sequentially combining the layers. Precisely, for $x \in \mathcal{X}$,

$$NN(x) = (L_d \circ L_{d-1} \circ \cdots L_2 \circ L_1)(x).$$

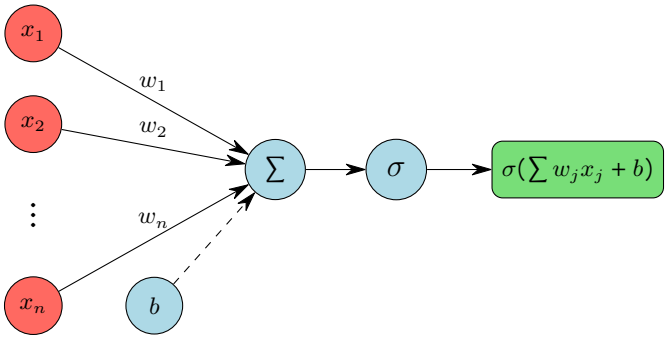

Figure 2.4: An artificial neuron, the basic unit of a neural network.

In other words, a feedforward neural network consists of an alternating sequence of linear transformations and activation functions. The latter, in particular, are responsible for injecting nonlinearity into the network, hence they crucially contribute to the potential of these models: without the action of the these functions, a neural network would simply be a linear model. Many types of activation functions are available, such as the *rectifying linear unit* (ReLU) and its variants, the *tanh* or the *sigmoid*; for a survey on the different activation functions, we refer to [Dat20]. Clearly, choosing an activation function, suitable for the problem to solve, represents an important decision for the success of the model.

Now, training a neural network is, conceptually, not different from training other types of machine learning models, that is, we need to apply the learning paradigm described in Section 2.1: define a model space, choose a loss function to compute the error, design an optimization procedure to minimize this error over the data. Differently from other models, however, in deep learning algorithms, the error function is usually non-convex, due to the nonlinearity of the model, and the solution space is extremely high-dimensional, due to the complexity of the network. In other words, training a network is a hard problem; hence, for these models, we usually renounce to optimality, and rather adopt heuristic methods that simply drive the error to a very low value, by searching for a good, yet suboptimal, solution.
One of the most popular class of optimizers for neural networks is represented by the family of *gradient descent* algorithms, whose description, however, is out of the scope of this thesis. A detailed explanation of the gradient descent is given in the textbook of Goodfellow at al. [GBC16], while a very clear overview of the different types of algorithms within this family is provided by Ruder [Rud17].

Important hyperparameters of a deep learning algorithms are the ones defining the architecture of the network, namely, the number of hidden layers and the number of hidden units for each layer. The values of these hyperparameters, as for other learning models, are commonly set by means of cross-validation.

For a comprehensive study of neural network models and deep learning algorithms, we refer again to Goodfellow at al. [GBC16].

Chapter 3

A Machine Learning Strategy for the Use of Local Cuts

In the first section of this chapter we describe and motivate the central challenge of this thesis, that is, the development of an efficient strategy to decide whether to use local cutting planes while solving a mixed-integer program. In particular, we observe how naturally a machine learning approach suits the goal that we want to achieve. Hence, in the second section, we survey the major developments provided to the young and exciting field of research to which this thesis aims to contribute, namely, the integration of machine learning and combinatorial optimization; in particular, we discuss the different approaches, proposed since the beginning of this research area, to use the former in the direction of the latter. Finally, in the third section, we provide a theoretical description of the method that we adopt to leverage machine learning for solving the decision problem mentioned above.

3.1 A Crucial Decision: Cut vs Not Cut

It is undoubted that the integration of cutting planes into the branch-and-bound main framework, firstly suggested during the 80s [HCP83, VRW87], then formalized during the 90s [PR91, BCC96, BCCN96], has represented a breakthrough in MIP computation and played a central role in the move to the current generation of MIP solving systems. The great impact that cutting planes have had on the evolution of MIP solvers emerges, for example, from the experiment conducted by Achterberg and Bixby [AB08], where the authors compared the MIP-solving performance of all Cplex versions from release 1.2, the first with MIP capabilities, up to release 11, the newest at that time. Precisely, the authors reported a speed-up of 22.3 from Cplex 6.0 to Cplex 6.5, the first release with full cutting plane capability. In particular, the improvement of the software between these two versions represents the biggest step forward in the version-to-version scale reported in the paper, and was mostly provided by the introduction of cutting planes.

We have seen in Section 1.3.1 , however, that there are many possible ways to use

cutting planes into the B&B framework, each one determined by a precise combination of different algorithmic choices. Unsurprisingly, not all combinations give rise to a cutting strategy that is really effective and efficient for the problem to solve. One of the first decisions that need to be taken, while implementing the integration of cuts into the B&B, is whether the cutting activity should be limited to the root node, or performed during the whole search, that is, whether to apply only global cuts, or to use both global and local ones (see Section 1.3.1).
In general, the impact that a cutting activity has on the running time of the solver results in the sum of two opposite weights. On one side, adding cutting planes directly improves the dual bound, hence, as a side effect, it provides a reduction of the number of nodes to be processed; on the other side, the use of cuts requires a non-ignorable computational effort, both for generating them and for re-solving the tightened problem (see Section 1.3). Now, it is known that cutting planes are particularly powerful at superficial levels, especially at the root node [AW13], while their effectiveness tends to deteriorate together with the growth of the tree depth. This means that, while at the beginning of the search the benefits provided by the use of cutting planes, for the great majority of problems, outweigh the drawbacks, this trend progressively changes as the search dives deeper, until a certain point from which it starts to reverse. This would suggest, as a reasonable criterion, the one of choosing, for the given input instance, a maximum depth level k, after which the cutting procedure should be stopped. There is, however, another consideration that should be made: the use of local cuts, which are valid, in general, only locally [Pad05], is incompatible with conflict analysis, a technique, widely used in today MIP solvers, that consists in exploiting the information deduced by the infeasibility of a subproblem, when this is encountered during the B&B search, in order to produce a new constraint for the global problem [Ach07a, WBH21]. Indeed, if the reason for the node's infeasibility comprises some constraints that are valid only locally, the constraint deduced by analyzing this node will be valid, in general, only locally as well. In other words, the use of cuts at internal nodes prevents the use of conflict analysis, and preferring the former routine over the latter is not always the most convenient choice. For a more detailed discussion on the compatibility between these two techniques, we refer to the paper of Berthold and Witzig [WB21]. Hence, this last observation seems to reduce the more general question, asking for the optimal value of k, to the more specific (binary) one, that is, whether to cut only at $k = 0$, or to keep cutting even for $k > 0$; equivalently, whether local cuts, for the given input problem, should be used or not: *local-cut vs no-local-cut*. Studying binary decision problems is, in particular, a common trend in MIP research, as exemplified by [KLP17, BH21, BLZ18].

Now, up until the time of writing this thesis, no efficient criterion has been developed, at least to the best of the author's knowledge, to provide an answer to this question.
In 1997, Cordier *et al.* conducted a computational experiment to evaluate the relative performance between the local-cut approach and the no-local-cut one [CMLW97]. Precisely, the authors run their solver, called BC-OPT and based on the XPRESS-MP system [Xpr], over the instances of the MIPLIB3.0 library [BCMS90] twice, once by disabling the cutting procedure after the first branching, the other by keeping generating cuts every

8 levels of the search tree. From the results reported in the paper, the authors drew the following conclusion: even though there are few instances, among the ones labeled as hard for the solver (i.e. the ones solved in more than 5 minutes), on which the local-cut setting outperforms the no-local-cut one, for most of the instances, the no-local-cut method is 10 to 20% faster than its competitor.
For more recent solvers, however, the scientific literature and the experience of the author seem to contradict the conclusion provided in the paper just mentioned [CMLW97]. Indeed, for more advanced solvers, which are clearly endowed with more complete cutting plane techniques and, overall, with a more mature algorithmic framework, the local-cut approach seems to dominate the no-local-cut one.
An example is provided by the experiment, conducted by Achterberg and Wunderling in 2013 [AW13], to measure the impact of each class of cutting planes in CPLEX 12.5, over a set of MIP instances coming from a mix of publicly available and commercial sources. Precisely, the authors of the paper compared the default configuration of the solver, consisting in the use of different classes of cutting planes both at the root node and internally, against turning off each cutting plane separator individually and, finally, against the cut-and-branch setting, which instead makes only use of global cuts. Among the different conclusions drawn by the authors, the one, interesting for this thesis, is that the performance of the solver, when local cuts are disabled, degrades by 23%, hence declaring the local-cut setup, on average, more efficient than the other.
To give a more direct flavor of the relative performance between the local-cut and the no-local-cut method, we present a study, conducted by the author of this paper, in which the two approaches are compared on a test bed consisting of 6 permutations of each problem of the Benchmark Set from MIPLIB 2017 [GHG+21], for a total of 1155 instances [1]. Precisely, the experiment consists in running FICO Xpress 8.9 [Xpr], up to a time limit of $7200s$, on each instance of the set twice, once by cutting only at the root node, the other by also using a local cutting procedure, as implemented in the default version of the solver (note that, for this experiment, we use the same computational setup adopted for the ones presented in Chapter 5, and it is described in detail in Section 5.1).
A first evaluation of the relative performance between the two competitors can be made by partitioning the dataset into four parts, according to whether an instance is solved or unsolved (within the time limit) by each of the two methods. Table 3.1 reports the number of data points across the partition, where the rows refer to the local-cut approach $\mathcal{C}$, while the columns to the no-local-cut one $\mathcal{NC}$.

As clearly shown by the table, the highest percentage of instances is given by the ones that are either solved or unsolved by both methods, given that these two sets represent, together, 88% of the entire test set. Now, of the remaining instances, $4/5$ could be solved by $\mathcal{C}$ but not by $\mathcal{NC}$, meaning that, by simply counting the number of solved instances, local-cut is 4 times better than no-local-cut. As a second evaluation, we count the number of instances on which one method is *at least 10%* faster than the other, hence we exclude all those instances on which the two approaches are roughly equivalent. It

[1] The Benchmark Set comprises 240 problems; the 1400 instances in our original dataset, however, are reduced to 1155 by a data cleaning process (see Section 5.1).

$\mathcal{C}$ \ $\mathcal{NC}$	*Solved*	*Unsolved*
Solved	747	107
Unsolved	28	273

Table 3.1: Number of instances according to whether they are solved by each of the two methods.

turns out that $\mathcal{C}$ outperforms $\mathcal{NC}$ on 477 of these instances, representing 41.3% of the entire dataset, while the opposite holds for 258 instances, constituting 22.3% of the same set; on the remaining 36.4% of the test set, instead, the two methods perform similarly. Finally, we compute the average running time (precisely the shifted geometric mean, defined in (4.3)) of the two approaches over the entire dataset. It results that $\mathcal{C}$ can solve a problem, on average, in 462 seconds, in contrast with 634 seconds of $\mathcal{NC}$, meaning that the former is 27% faster than the latter.
Hence, the results obtained by our study seem to confirm the conclusion drawn by Achterberg and Wunderling in the paper mentioned above [AW13]. In other words, if we had to decide, between the two methods, which one to use in FICO Xpress by default, then we would definitely choose the local-cut one. The interesting observation, however, is that the instances that could be solved within the time limit only by $\mathcal{NC}$, as well as the ones on which $\mathcal{NC}$ is faster than $\mathcal{C}$, represent a relevant percentage of the dataset. That is, if we had an efficient strategy to detect whether the input problem is one of those problems on which $\mathcal{NC}$ works better than $\mathcal{C}$, hence to solve this problem by using the no-local-cut method instead of the other, then we could solve 2.4% of instances more within the time limit, and decrease the average running time, over the considered dataset, to 410 seconds, hence improving the performance of the solver by 11%.

In conclusion, the entire discussion provided above leads to the following question: given a MIP instance P, should we use local cuts while solving P?
Developing an efficient strategy to make this choice represents the central challenge of this thesis. This is, in particular, where machine learning comes into play: given that, as far as the author knows, a deep mathematical comprehension of this decision problem is still missing, machine learning looks like a natural candidate to fill this gap of knowledge, hence to take such a choice in a principled and structured way.

3.2 Machine Learning for Combinatorial Optimization

Machine Learning (ML) and Mathematical Optimization (MO) have had a long and fruitful relationship. On one side, indeed, MO has always lied at the heart of ML and strongly contributed to its success: optimization techniques are employed in ML not only in the

training process, which is formulated, in most of the cases, as an optimization problem (see Chapter 2), but also in many other phases of the entire learning cycle, such as data preprocessing [TGH11], hyperparameter tuning [BB12] or architecture design in deep learning [FJ18]. A survey of common optimization techniques from a machine learning perspective can be found in [GGNS21, SCZZ20]. On the other side, the interplay between the two disciplines has been fertile, especially in recent years, also in the other direction, with ML tools being used in different MO applications [DHS11, MK11, CdAMJ17]. The Combinatorial Optimization (CO) community, in particular, has witnessed, during the last decade, a rapid growth of interest in the use of learning techniques to improve and complement the existing algorithms. This innovative and prolific area of research is commonly referred to as *Machine Learning for Combinatorial Optimization*, and represents the domain to which this thesis aims to belong and to contribute.

The general idea behind this line of research consists in employing ML tools to enhance, or replace, all those hand-crafted heuristics on which state-of-the-art algorithms heavily rely to make crucial decisions. In particular, in this context, two reasons essentially motivate the efforts for integrating ML routines into optimization frameworks. One case is when the available solving strategies for the decision problem are effective, but inefficient, that is, they provide acceptable solutions but require a prohibitive computational cost. ML, in this case, can help in replacing the heavy computations with a fast approximation. The other case is when the problem is still missing a rigorous characterization, that is, no mathematical knowledge of the problem is available yet. ML, in this second case, can be exploited to produce valid policies for driving the decision process. Now, in their paper from 2021, "Machine Learning for Combinatorial Optimization: a Methodological Tour d'Horizon" [BLP21], Bengio, Lodi and Prouvost identify three general paradigms to integrate ML into CO algorithms.
The first of them consists in completely bypassing the entire optimization algorithm (Figure 3.1). Precisely, the idea is to learn an ML model directly from the instance space to the solution space, this is why this approach is referred to as *End-to-End Learning*. Notable contributions, all employing deep learning techniques, have been given to this class of methods in the context of stochastic optimization [LLB$^+$21], semidefinite programming [BLBMT18] and, especially, the Traveling Salesperson Problem (TSP) [VFJ17, BPL$^+$17, ER18, NVBB18, KvHW19]. In mixed-integer programming, in particular, a remarkable advancement in the use of the End-to-End Learning has been provided recently in [NBG$^+$20], where the authors use this paradigm to develop an ML solver capable of handling not only specific classes of MIPs, but a more diverse range of real-world problems, such as the ones contained in MIPLIB2017 [GHG$^+$21] and the ones coming from several Google's internal datasets.

A more involving integration of ML into CO frameworks consists in the so-called *Learning Properties*, where the ML model is used to select, before starting the actual solve of the input problem, a suitable configuration of the solving process, hence improving the CO algorithm rather than replacing it (Figure 3.2). The execution of complex optimization algorithms, indeed, is controlled by a (usually very large) set of parameters

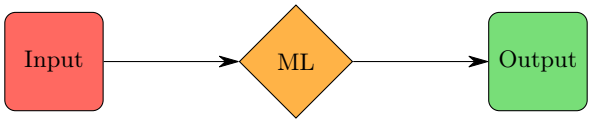

Figure 3.1: The ML model completely replaces the CO algorithm.

(the ones called hyperparameters in ML), whose setup needs to be chosen carefully according to the given problem, being it crucially responsible for the success of the solve. For many of these parameters, however, the dependencies between the space of possible values that they can take and the space of MIP problems are not clear yet. ML, in this context, looks like a natural candidate to fill this gap of knowledge, that is, to detect patterns within the instance space and discover new policies to provide a better parametrization of the solver. For example, in mixed-integer programming, ML ca be used to estimate, beforehand, whether a Dantzig-Wolf decomposition, for a linear program, would be able to make the solver faster [KLP17], to predict which scaling method, again for the linear case, is the most reliable one [BH21], as well as to decide, for a quadratic program, if the linearization procedure would be convenient [BLZ18]. Other applications of this paradigm comprises the automatic generation of a metaheuristic in Bipartite Boolean Quadratic Programming [KPP17], or the automatic configuration of the search for the best algorithm in Genetic Programming [MLIDLS14]. We observe that, in the Learning Properties paradigm, the information that the ML agent uses to take its decision is not limited to the "static" features of the problem, such as its combinatorial structure, but might comprise also some "dynamic" features, that is, statistics provided by the solver itself while running on the problem, for example the decomposition statistics used in [KLP17], or the LP information in [BLZ18].

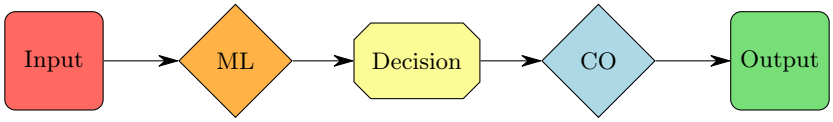

Figure 3.2: The ML model provides a valuable piece of information to the CO algorithm.

Finally, the third paradigm to combine ML and CO can be referred to as *Learning Repeated Decisions*, and consists in a full integration of the learned model into the optimization framework (Figure 3.3). Precisely, in this case, the high-level optimization algorithm, while running, repeatedly queries the same ML subroutine to make the same type of decision. The ML model, in other words, is embedded into the optimization framework to assist it along the entire process. The repetitive interaction between ML and CO makes this paradigm particularly suitable for solving all those decision problems that usually recur during the execution of CO algorithms, such as variable selection [LZ17, GGK$^+$20], cut selection [TAF20b, HWL$^+$21], or heuristic selection [Hen18, HMW19]. The input received by the ML agent, in this case, consists of some description of the algorithm

state, which might also include the definition of the problem.

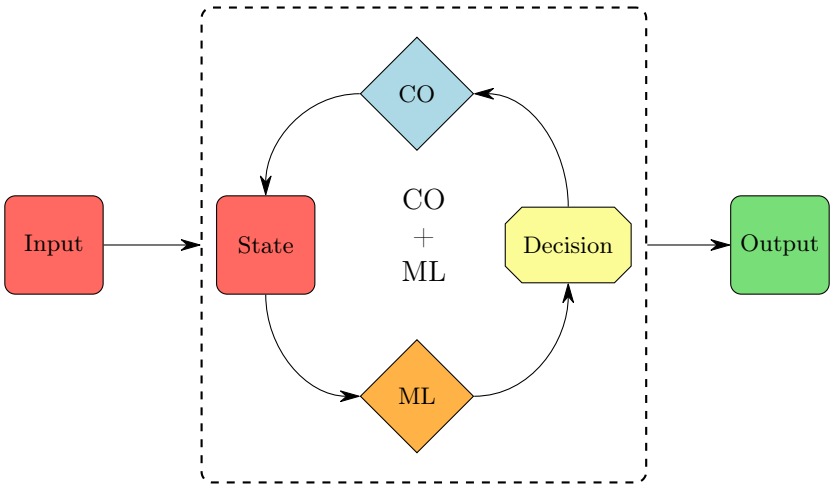

Figure 3.3: The ML model assists the CO algorithm along the entire process.

Now, the Learning Properties paradigm, among the three described above, looks like the most natural candidate for the goal that we aim to achieve in this thesis, that is, developing an efficient policy to decide whether to use local cuts while solving mixed integer programs. The idea, indeed, is to learn the mathematical relationship between the space of MIP instances and the space of possible answers to our question, i.e., *local-cut* or *no-local-cut*, in the hope of remedying the lack of comprehension that, to the best of the author's knowledge, the CO community is still suffering around this question. In the following section, in particular, we provide a theoretical explanation of our solution to the problem, while in the next chapter we describe the adopted methodology in detail. To conclude this section, we clarify that the three paradigms described here represent only general methodologies to make ML and CO work together, and they are neither meant to be exhaustive nor disjoint, but just a natural way to look at the literature.

3.3 Learning to Use Local Cuts

The central challenge of this thesis, as discussed in Section 3.1, is to develop a policy to decide whether, for each input MIP problem, local cuts should be used or not, hence to employ this policy within the solving process to speed up the running time of the solver. The software at the basis of our research and methodology is the state-of-the-art solver FICO Xpress [Xpr].
The approach that we use to achieve our goal consists in considering generic optimization problems as data points, hence in using machine learning techniques to inquire the relevant distribution of problems to use for learning on our binary classification task. In particular, a MIP instance is represented by a n-dimensional vector of features, each

one describing a particular property, or characteristic, of the problem. The domain of possible answers to our question is $S = \{\mathcal{C}, \mathcal{NC}\}$, with $\mathcal{C}$ and $\mathcal{NC}$ representing, as in Section 3.1, the local-cut method and the no-local-cut one, respectively. Our ultimate goal is to produce a binary classifier $M : \mathbb{R}^n \to S$ that, hopefully, is able to predict, for each given problem, whether $\mathcal{C}$ is faster than $\mathcal{NC}$ or vice versa, hence to use this classifier as a subroutine of the solver. In other words, we are, evidently, in the context of the Learning Properties paradigm (Figure 3.2), that is, we want to use an ML agent to configure, before the actual solve, the parameter that allows to activate or deactivate the local cutting procedure.
In particular, the ML subroutine is inserted into the workflow of the solver after the presolving process and the global cutting loop, but before the first branching, as illustrated by the flowchart in Figure 3.4. Our decision, indeed, does not need to be taken right at the beginning of the solve, but can wait until the moment in which the search leaves the root node and starts to explore the tree internally. This allows us to use, for our choice, not only the problem itself, such as its combinatorial structure, but also the information produced by the solver during all the process preceding the first branching.

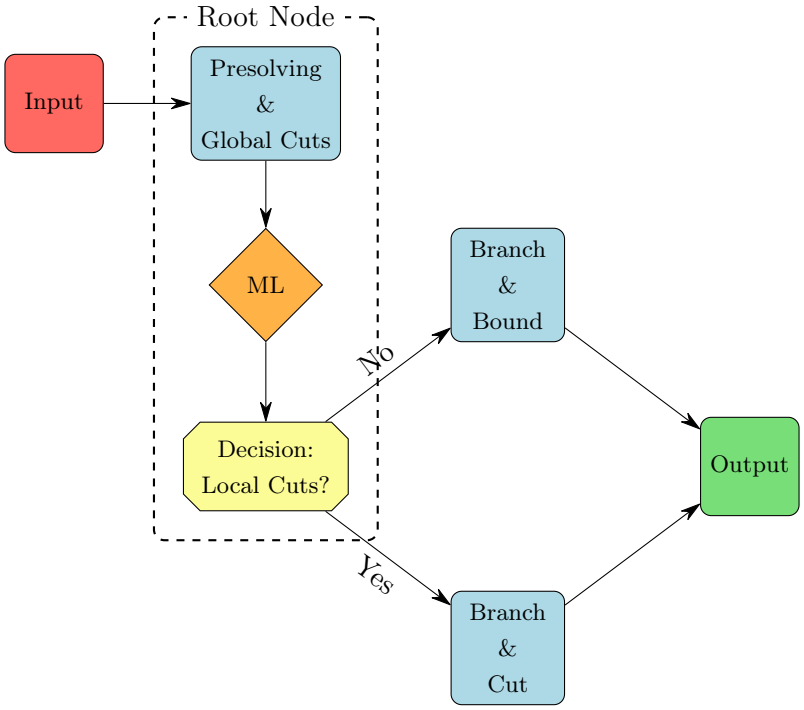

Figure 3.4: The ML agent is responsible for deciding, at the end of the root node, whether the cut generator should be deactivated or not.

Hence, right before leaving the root node, the ML agent looks at the state of the problem and makes its prediction, according to which the parameter for the local cutting procedure is configured. Precisely, if the model predicts that $\mathcal{C}$ will be faster than

$\mathcal{NC}$, the solve proceeds with the branch-and-cut scheme, while in the other case, the solve continues with the branch-and-bound one. We clarify, however, that we are only interested in deciding whether cutting planes should be used locally or not, hence in impacting the behavior of the solve only for this specific aspect. We rely instead on the default configuration of Xpress for all other decisions required by the solve, hence for the implementation of the usual subroutines that take part in the solving process, such as presolving techniques, primal heuristics, conflict analysis, the global cutting loop as well as the local cutting procedure, when $\mathcal{C}$ is chosen to be the solving method. In other words, both the methods $\mathcal{NC}$ and $\mathcal{C}$ consist in the default run of Xpress, they only differ from each other in the fact that, in the former, cutting planes are deactivated before the first branching, while in the latter, the cut generator is kept activated also at internal nodes. Precisely, all the parameters that control the local cutting procedure executed in the $\mathcal{C}$ method, such as the maximum tree depth at which the cutting activity is stopped, the types of cuts that are generated, the frequency at which cuts are added during the search and, in general, the aggressiveness of the cutting routine (see Section 1.3.1), are left, again, in their default configuration.

In a supervised learning framework, we train and test 3 ML models, a Linear Model (`LM`), a Random Forest (`RF`) and a Neural Network (`NN`), each one providing a different policy for our decision. The rigorous formulation of the learning problem, as well as the methodology adopted in our approach, are described, in detail, in the following chapter.

Chapter 4

Methodological Approach

In this chapter, we provide a detailed description of the methodological approach that we adopt to answer the question stated in Chapter 3. In particular, we subdivide our approach into 4 steps. In the first two phases, we design the features and define the labels, hence we frame our decision problem as a supervised learning task (see Chapter 2). In the following two steps, instead, we describe the process that we run to construct a suitable dataset for our ML application, hence we explain how, on this set, we develop and evaluate our solution to the problem. The proposed methodology can be implemented for any MIP solver $\mathcal{S}$ (assuming that it supports both the $\mathcal{C}$ and the $\mathcal{NC}$ configuration) and any problem set $\mathcal{P} \subseteq \mathcal{P}$, where $\mathcal{P}$ denotes the space of all MIP problems. Note that, although in this thesis we make use of different solvers and problem sets, most of our computational study is developed by using the Benchmark Set from MIPLIB 2017 [GHG+21], denoted as *Miplib17*, and FICO Xpress 8.9, denoted as *Xpr8.9*, hence we consider this setting as our default one.

4.1 Feature Design

Features, in machine learning, are independent variables providing a representation of the input instance that is fed into the ML model. In other words, they constitute the information which the model looks at to make its prediction (see Section 2.1).

The objects that we need to represent, in our case, are mixed-integer programs. To this aim, for the given solver $\mathcal{S}$, we describe a problem $p \in \mathcal{P}$, both in its mathematical formulation and in its computational behavior, by means of 32 features $x_1, \dots, x_{32}$, that should condense the relevant pieces of information leading to an algorithmic discrimination between $\mathcal{C}$ and $\mathcal{NC}$. The resulting *feature space* is denoted by $\mathcal{F}_{\mathcal{S}} \subseteq \mathbb{R}^{32}$, and the *feature vector* of p by $x^p_{\mathcal{S}} \in \mathcal{F}_{\mathcal{S}}$.
A comprehensive description of our feature space $\mathcal{F}_{\mathcal{S}}$, together with the definition of each individual feature, is provided in Table 4.1, where we adopt the following notation. Given a MIP problem $p \in \mathcal{P}$, we denote by m the number of its rows and by n the number of its columns, hence by $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$ and $c \in \mathbb{R}^n$ its constraint matrix, right-hand side

vector and objective vector, respectively. Moreover, we denote by A' the set of absolute values of the non-zero entries of the matrix A, and similarly for the vectors b and c. Precisely,

$$A' = \{|A_{i,j}| \text{ s.t. } A_{i,j} \neq 0\}_{(i,j)\in[m]\times[n]},$$
$$b' = \{|b_i| \text{ s.t. } b_i \neq 0\}_{i\in[m]},$$
$$c' = \{|c_j| \text{ s.t. } c_j \neq 0\}_{j\in[n]}.$$

The number of remaining rows and columns after the presolving process, instead, are denoted by $\tilde{m}$ and $\tilde{n}$, respectively. Finally, we consider the following objective values at the root node: the objective value at the initial relaxation (Initial Bound), at the end of the cutting loop (Dual Bound) and at the incumbent (Primal Bound), if already available, and we indicate them as α, β, and γ, respectively. In particular, to better describe $\mathcal{F}_{\mathcal{S}}$, we define a double categorization of the features adopted. Firstly, we classify each of them as either *Static* or *Dynamic*: the former represent the problem in its mathematical formulation and combinatorial structure, hence they are completely solver-independent, the latter, instead, describe the computational properties of the problem, hence they are clearly solver-dependent. Moreover, we divide the static features into 4 groups, corresponding to different components of the problem formulation, such as *Basic*, *Variables*, *Constraints* and *Numerics*. More precisely, the first group provides some basic statistics about the problem, such as the size and the sparsity of its matrix, as well as the presence of eventual symmetries. The second and third group, instead, describe the variable composition and the constraint composition of the problem, respectively, hence provide a representation of its combinatorial structure. For the definition of each constraint class listed in Table 4.1, we refer to the MIPLIB 2017 website [Mip]. Finally, the last group of static features describes the order of magnitude of the problem data in the constraint matrix, right-hand side vector and objective vector. The dynamic features, instead, are split into 2 groups, corresponding to the different stages of the problem solve, that is, *Preprocessing* and *Global_ Cuts*, in which the solve is not yet affected by our decision. Precisely, the former group describes the effect of the presolver on the problem, by providing some basic information about the presolved matrix, while the latter quantifies the impact of the global cutting loop, by measuring the gaps at the root node among the different objective values, as well as the *gap closure*, a commonly used metric to evaluate the effectiveness of cutting planes.

To conclude this section, we observe that all features defined in Table 4.1 are continuous variables, except the one that describes the symmetries of the problem, x_{23}, which is instead a binary variable. Moreover, we clarify that, for our learning experiments, we might not use the precise feature values as they are defined in Table 4.1, since we might need to rescale and normalize them across the particular dataset adopted. This is a common practice and a necessary step in every machine learning application, generally performed during the data preprocessing phase. A description of the transformations that we use to preprocess our data is provided in Section 5.1.

	Feature	Definition	Feature	Definition
Static			~ *Basic* ~	
	x_1 `Rows`	$\ln(m)$	x_3 `NonZeros`	ratio of non-zeros, over matrix size $m \times n$
	x_2 `Columns`	$\ln(n)$	x_4 `Symmetries`	1 if any simmetry, 0 otherwise
			~ *Variables* ~	
	x_5 `Binaries`	ratio of binary variables, over n	x_6 `Integers`	ratio of integer variables, over n
			~ *Constraints* ~	
	x_7 `LessThan_Const`	Ratio of constraints per constraint type, over m	x_{15} `Knapsack_Const`	Ratio of constraints per constraint type, over m
	x_8 `GreaterThan_Const`		x_{16} `KnapsackInteger_Const`	
	x_9 `Equality_Const`		x_{17} `BinaryPacking_Const`	
	x_{10} `SetPartitioning_Const`		x_{18} `VariableLowerBound_Const`	
	x_{11} `SetPacking_Const`		x_{19} `VariableUpperBound_Const`	
	x_{12} `SetCovering_Const`		x_{20} `MixedBinary_Const`	
	x_{13} `Cardinality_Const`		x_{21} `MixedInteger_Const`	
	x_{14} `KnapsackEquality_Const`		x_{22} `Continuous_Const`	
			~ *Numerics* ~	
	x_{23} `Coefficient_Oom`	$\ln(\max A' / \min A')$	x_{25} `Objective_Oom`	$\ln(\max c' / \min c')$
	x_{24} `RightHandSide_Oom`	$\ln(\max b' / \min b')$		
Dynamic			~ *Presolving* ~	
	x_{26} `PresolRows`	$\ln(\tilde{m})$	x_{28} `PresolIntegers`	ratio of presolved integer variables, over n
	x_{27} `PresolColumns`	$\ln(\tilde{n})$		
			~ *GlobalCuts* ~	
	x_{29} `DualInitial_Gap`	$\lvert\beta - \alpha\rvert / \max(\lvert\beta\rvert, \lvert\alpha\rvert, \lvert\beta - \alpha\rvert)$	x_{31} `PrimalInitial_Gap`	$\lvert\gamma - \alpha\rvert / \max(\lvert\gamma\rvert, \lvert\alpha\rvert, \lvert\gamma - \alpha\rvert)$
	x_{30} `PrimalDual_Gap`	$\lvert\gamma - \beta\rvert / \max(\lvert\gamma\rvert, \lvert\beta\rvert, \lvert\gamma - \beta\rvert)$	x_{32} `Gap_Closure`	$1 - x_{29}/x_{31}$

Table 4.1: A prospect of the feature space $\mathcal{F}_{\mathcal{S}}$

4.2 Label Definition

Labels, in supervised machine learning, are the target values that the ML model, by looking at the features of the input problems, tries to predict. They provide, during the training phase, the pieces of information necessary to supervise the learning model, hence to correct its behavior and to drive it towards an acceptable performance (see Chapter 2).

In our case, the labels should communicate to the learner which one of the two methods, between $\mathcal{C}$ and $\mathcal{NC}$, is the more efficient one, for all MIP instances in our dataset. Now, even though our final model should work as a binary classifier, we actually train it as a regressor, hence we design our output space as a continuous set, rather than a binary one.
For a MIP problem $p \in \mathcal{P}$ and a MIP solver $\mathcal{S}$, there are different pieces of information that we could use to quantify the performance of $\mathcal{S}$ in solving p with the two configurations $\mathcal{C}$ and $\mathcal{NC}$, such as the number of tree nodes explored or the peak amount of memory consumed, the time to optimality or the *primal-dual integral* [Ber13]. However, through our model, we are mostly interested in improving the solver in terms of running time; this is why, in defining the labels that supervise the training process, we take into account solely the time to optimality, while we use the other measurements in the testing phase, in order to compare the produced models in terms of different metrics, hence to evaluate them more exhaustively. Precisely, we define the labels as follows.

Definition 4.1. *Given a solver $\mathcal{S}$ and a problem $p \in \mathcal{P}$, let $Time_\phi(p)$ be the running time of $\mathcal{S}$ while solving p with configuration $\phi \in \{\mathcal{C}, \mathcal{NC}\}$.*
We define the label of p, and we denote it by $y^p_{\mathcal{S}}$, as the speedup factor between $Time_{\mathcal{C}}(p)$ and $Time_{\mathcal{NC}}(p)$, rescaled by means of the logarithmic function log_2. Precisely,

$$y^p_{\mathcal{S}} = Speedup(p) = log_2\left(\frac{Time_{\mathcal{C}}(p) + 1}{Time_{\mathcal{NC}}(p) + 1}\right) \in \mathbb{R}.$$

Note that the runtimes of the two methods are both augmented by 1, to mitigate the impact of very small numbers, as well as to prevent the division by zero. We observe that the speedup is negative when $\mathcal{C}$ is faster than $\mathcal{NC}$, that is, when $T_{\mathcal{C}} < T_{\mathcal{NC}}$, it is positive otherwise; when the speedup is 0 (or close to 0), then the two methods are equivalent (or roughly equivalent).

Figure 4.1 depicts the distribution of the speedup factors, for our default solver *Xpr8.9* [1], over our default problem set *Miplib17** (that will be defined in Section 4.3). The two-color gradient scale, used to fill the histogram, reflects the continuous speedup values over the instances of our dataset, with the nuances of red and blue indicating $\mathcal{C}$ and $\mathcal{NC}$, respectively. In other words, the colors of the bins varies continuously, from left to right, between plain red, representing those instances where $\mathcal{C}$ is substantially better than $\mathcal{NC}$, to plain blue, corresponding to the opposite situation. Around 0, instead, the bins are

[1] Each solve is executed up to a time limit $T = 7200s$, within the computational environment described in Section 5.1

filled with the tone of green that derives from the interpolation between the $\mathcal{C}$- and the $\mathcal{NC}$-color, hence indicating that, on those instances, the competition between the two methods results in a tie.

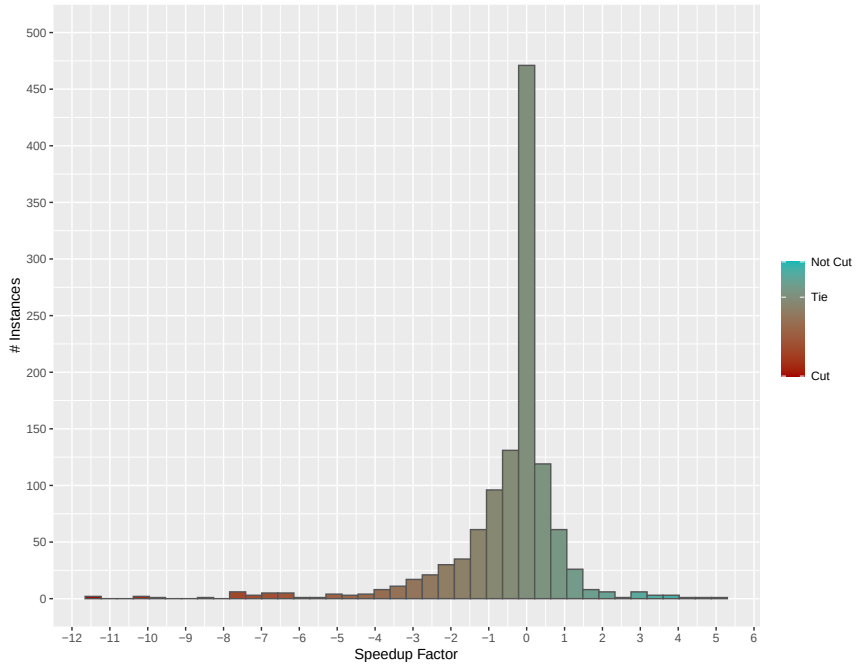

Figure 4.1: distribution of the speedup factors over *Miplib17**, computed with *Xpr8.9* and colored continuously according to the speedup value.

Figure 4.2, instead, displays the runtime of $\mathcal{C}$ and $\mathcal{NC}$ plotted against each other. In other words, the points of the scatter plot correspond to the pairs $(Time_{\mathcal{C}}(p_s), Time_{\mathcal{NC}}(p_s))$, obtained by running *Xpr8.9* [1] over each instance $p_s \in$ *Miplib17** (that will be defined in Section 4.3). Again, the color of the points reflects the relative performance of the two methods, hence it varies continuously, from the upper-left corner of the plot to the lower-right one, between plain red, representing those instances on which $\mathcal{C}$ and $\mathcal{NC}$ reach their minimum and maximum runtime, respectively, and plain blue, corresponding to the opposite situation. Along the diagonal $y = x$, instead, the color of the points results from the interpolation of the two colors at the extremes, indicating that, on these points, the two methods perform similarly.

Now, with the features designed in Section 4.1, and the labels defined in this section, we can formulate our decision problem as a regression problem, where the task is to produce, for a solver $\mathcal{S}$, a model

$$M_{\mathcal{S}} : \mathcal{F}_{\mathcal{S}} \longrightarrow \mathbb{R}, \tag{4.1}$$

that for each input vector $x^p_{\mathcal{S}}$ approximates the speedup factor $y^p_{\mathcal{S}}$ between the two runs,

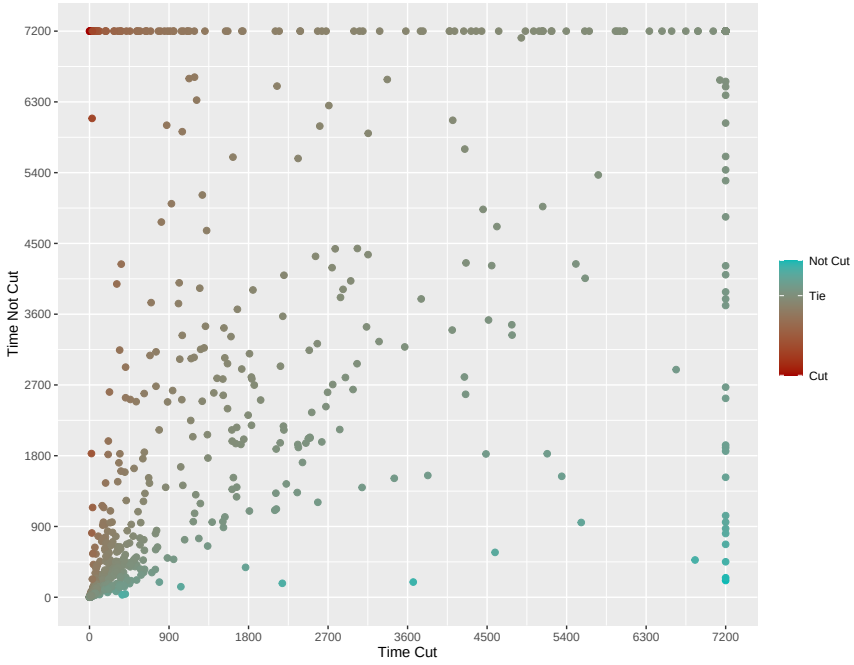

Figure 4.2: instance-by-instance comparison between the runtimes of the two methods over *Miplib17**, computed with *Xpr8.9* and colored continuously according to the speedup value.

$\mathcal{C}$ and $\mathcal{NC}$, of the solver $\mathcal{S}$ over $p \in \mathcal{P}$, that is,

$$M(x^p_{\mathcal{S}}) \approx y^p_{\mathcal{S}}.$$

In conclusion of this section, we observe that, in preliminary phases of our study, we also considered different ML formulations for our decision problem, other than the one described here. In particular, we substituted the speedup factor of the running time with the speedup factor of the primal-dual integral, as well as we learned the classes $\mathcal{C}$ and $\mathcal{NC}$ directly, hence switching from a regression framework to a classification one. In particular, the results obtained from the latter approach turned out to be particularly relevant, this is why we decided to report them in Section 5.4. However, the best results could be observed by using the labels and the problem formulation illustrated in this section; this is why, in this thesis, we consider this methodology as the default one.

4.3 Data Collection

Data represents the foundation of machine learning and the primary resource consumed by any learning algorithm. This is why *data collection*, i.e., the process of gathering meaningful information for the learning goals, represents a building block of any ML application. In what follows, we illustrate the procedure that we use to gather the raw data,

on which applying our methodology and conducting our computational experiments.

Precisely, for the given solver $\mathcal{S}$ and problem set $\mathcal{P}$, the procedure of gathering such data can be described as follows. Firstly, we apply 6 random permutations to the elements of $\mathcal{P}$, each one associated to a seed $s = 0, \ldots, 5$ (with 0 corresponding to the identity permutation), in order to enlarge and diversify our ground set of problems. We denote the expanded set by $\mathcal{P}^*$, and each of its elements by p_s, which we refer to as an *instance* of our problem set, where $p \in \mathcal{P}$ and $s = 0, \ldots, 5$. Note that the 6 instances produced from p, $p_0, \ldots, p_5$, although mathematically equivalent, can have a very different computational behavior, as shown in [LT14]. Then, from each instance p_s, we extract all necessary data that we need to conduct our research. In particular, we start by collecting, from p_s, the solver-independent information that we use to describe the formulation of the problem and its mathematical properties, necessary for computing the static features. Then, we run the solver $\mathcal{S}$ over p_s twice, once with the local-cut configuration $\mathcal{C}$, the other with the no-local-cut one $\mathcal{NC}$. From these runs, in particular, we retain the relevant data, this time solver-dependent, that we use to describe the computational behavior of the problem at the root node, where the solve is not yet influenced by our choice, necessary for the dynamic features, as well as to describe the amount of resources consumed by each of the two approaches $\mathcal{C}$ and $\mathcal{NC}$, that we need for computing the labels and evaluating the learners .

Now, from the collected raw information, we compute the features and the labels for each instance p_s in our test bed $\mathcal{P}^*$, as described in Sections 4.1 and 4.2, respectively, hence we construct the set of features-label observations:

$$\mathcal{D}(\mathcal{S}, \mathcal{P}) = \{(x_{\mathcal{S}}^{p_s}, y_{\mathcal{S}}^{p_s}) : \ p_s \in \mathcal{P}^*\} \subseteq \mathcal{F}_{\mathcal{S}} \times \mathbb{R},$$

that we use as the ground dataset for the development of our data-driven approach. Note that, to simplify the notation, in the remaining of this chapter we refer to the dataset $\mathcal{D}(\mathcal{S}, \mathcal{P})$ simply as $\mathcal{D}$, and to the individual instances $(x_{\mathcal{S}}^{p_s}, y_{\mathcal{S}}^{p_s})$ simply as (x^{p_s}, x^{p_s}).

4.3.1 Data Split

As usual in machine learning, we do not use the entire dataset $\mathcal{D}$ for the training process, but from it, we detach a subset of data that we use, in the testing phase, to evaluate the generalization capabilities of the learned models (see Chapter 2). In our case, in designing a valid split criterion for $\mathcal{D}$, we take into account the particular structure of our ground problem set $\mathcal{P}^*$. In particular, we use the permuted problems of $\mathcal{P}$ (seeds $s = 1, \ldots, 5$) for the train set, while for the test set we use the original ones (seed $s = 0$). Precisely, let

$$\mathcal{P}^*_{train} = \{p_s \in \mathcal{P}^* : \ s = 1, \ldots, 5\} \quad \text{and} \quad \mathcal{P}^*_{test} = \{p_s \in \mathcal{P}^* : \ s = 0\}$$

be the set of *train instances* and *test instances*, respectively. Then, we define

$$\mathcal{D}_{train} = \{(x^{p_s}, y^{p_s}) : \ p_s \in \mathcal{P}^*_{train}\} \text{ and } \mathcal{D}_{test} = \{(x^{p_s}, y^{p_s}) : \ p_s \in \mathcal{P}^*_{test}\},$$

as our *train set* and *test set*, corresponding to roughly 83% and 17% of the dataset $\mathcal{D}$, respectively. Figure 4.3 represents, again, the scatter plot of the pairs of runtimes of the two competitors, computed with *Xpr8.9* [2] over *Miplib17**; this time, however, the points are colored by membership: in yellow the instances in $\mathcal{D}_{test}$, in green the ones in $\mathcal{D}_{train}$. We observe that, in preliminary experiments, we considered also different criteria for splitting our dataset; among the others, however, we found out that the data split described here is the one that guarantees the best results.

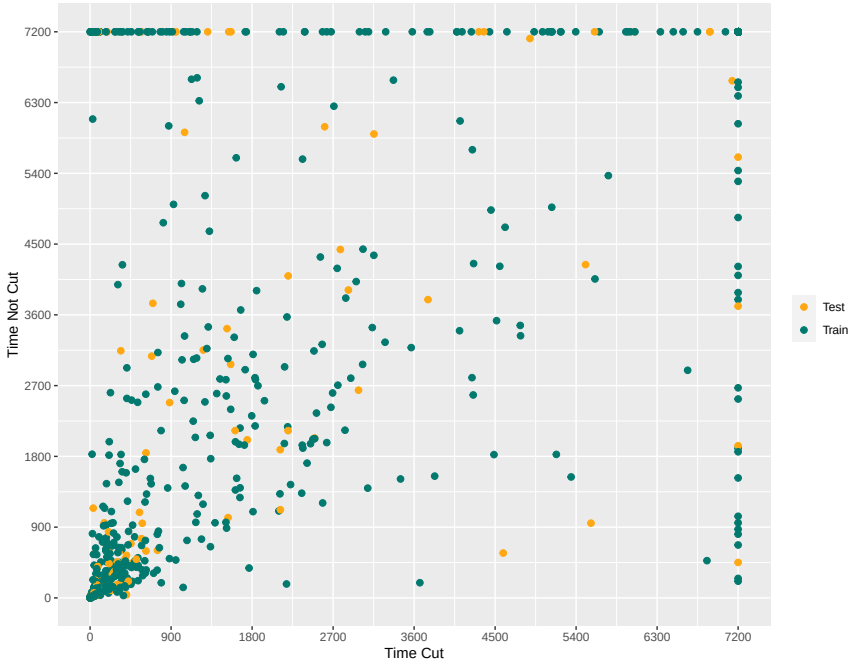

Figure 4.3: an instance-by-instance comparison between the runtimes of the two methods over *Miplib17**, computed with *Xpr8.9* and colored categorically according to the set membership.

4.4 Training & Testing

Model training is the core step of any machine learning application, and corresponds to the phase in which the actual model is built up from the available data. Model testing, instead, is the stage that immediately follows the training, and consists in evaluating the learned model on previously unknown data, in order to assess its generalization abilities, hence to check if it is ready to perform the desired task.

[2] Each solve is executed up to a time limit $T = 7200s$, within the computational environment described in Section 5.1

The model that we want to build is the predictor defined in (4.1). To this aim, we consider 3 machine learning models, a *linear model* (`LM`), a *random forest* (`RF`) and a *neural network* (`NN`), hence we train and test them over our dataset $\mathcal{D}$ as follows.

4.4.1 Training Methods

We perform the training of our models in the `R` programming language [R C13], where we encode our train set into an `R` data frame, with rows given by the entries of the set, while columns corresponding to the feature and label variables $x_1, \dots, x_{32}, y$. Within the `R` environment, in particular, we call this set `train.set`, and we specify the role of the variables, in the learning process, by defining a formula `f` through the following line of code:

$$\texttt{f} \leftarrow \texttt{y} \sim \texttt{x}_1 + \ldots + \texttt{x}_{32}.$$

Hence, we train our models as follows.

Linear Model: to train `LM`, we use the function `train`, provided by the add-on package `caret` [Kuh20], called with the parameter `method` set to `"lm"`. Precisely, we perform the training by running the following code snippet:

```
LM ← train(f, data = train.set, method = "lm").
```

Random Forest: similarly, we train `RF` by means of the `caret` function `train`, this time by setting the parameter `method` to `"rf"`. In this case, however, we also run a model validation process, in order to select a suitable number of decision trees to employ within the ensemble (see Section 2.3.2). Precisely, to tune this model's hyperparameter, we use the `trainControl` tool, again provided by `caret`, with `method` set to `"cv"` and `number` to `5`, in order to perform, over the train set, a 5-*fold cross-validation* with randomly generated folds (see Section 2.1.3). In other words, we learn `RF` by executing the following code snippet:

```
train.control ← trainControl(method = "cv", number = 5)
RF ← train(f, data = train.set, method = "rf",
           trControl = train.control).
```

The code above produces a random forest with 500 decision trees, which is the value selected by the model validation process for this hyperparameter.

Neural Network: for training `NN`, instead, we use the `neuralnet` function from the add-on package `neuralnet` [FGW19]. In particular, we train a neural network with a single hidden layer and 10 hidden units (see 2.4), and as stopping criteria for its training process, we set the function parameters `stepmax` to `1e7` and `threshold` to `0.2`, where `stepmax` represents the maximum number of iterations of the learning algorithm, while `threshold` provides a threshold for the partial derivatives of the error function. Precisely, we train `NN` by running the following code:

```
NN ← neuralnet(f, data = train.set, hidden = c(10),
               stepmax = 1e7, threshold = 0.2).
```

4.4.2 Testing Methods

Once that our models are trained from the data, they are ready to be tested, so that their quality can be evaluated. Now, from a machine learning perspective, one way to assess the performance of a learner, over a given observation (x, y), is to measure the *squared error* between the observed label y and the predicted label $M(x)$, that is, $(M(x) - y)^2$. The performance of the model can be then averaged over a certain test set $\mathcal{T}$ by computing the *Mean Squared Error* (*MSE*), in order to test the model on multiple instances, hence to obtain a more robust evaluation of its quality.

Definition 4.2. *Given a ML model M and a test set of observations $\mathcal{T}$, we define the Mean Squared Error of M over $\mathcal{T}$ as*

$$MSE(M) = \frac{1}{|\mathcal{T}|} \sum_{(x,y)\in\mathcal{T}} (M(x) - y)^2.$$

From an optimization point of view, however, we are mostly interested in quantifying the effectiveness of our model when this is deployed in practical MIP solving. This is why, in the testing phase, other than considering the standard ML metrics, we also, and especially, rely on those metrics that are commonly used to evaluate optimization algorithms, such as the running time (*Time*) and the primal-dual integral (*PDI*) [Ber13], the peak memory usage (*Memory*) and the number of explored nodes (*Nodes*), In order to capture the quality of the learner from the solver perspective, however, we firstly need to involve it, as a heuristic, in the solving process. To this aim, we need to convert the trained regressor, which predicts the relative performance between $\mathcal{C}$ and $\mathcal{NC}$, into a binary classifier, in order to make it able to directly discriminate between the two methods. Hence, we fix a *threshold* $\tau \in \mathbb{R}$ for the predicted speedup factor, and we use it as a switch between the two methods. Precisely, given a problem $p \in \mathcal{P}$, the method for solving p is chosen, according to the predicted speedup factor $M(x^p)$, as follows: if $M(x^p) \leq \tau$, we solve p with the local-cut configuration $\mathcal{C}$, otherwise, we use the no-local-cut one $\mathcal{NC}$, where x^p is the feature vector representing p (Figure 4.4).

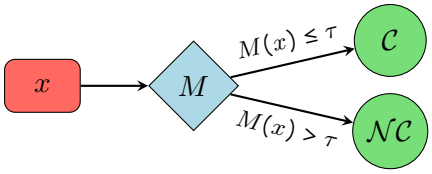

Figure 4.4: The decision process for the feature vector x.

Now, let $Pr_\phi(p)$ be the performance of the solver while running over the problem p with strategy $\phi \in \{\mathcal{C}, \mathcal{NC}\}$, where Pr is any of the optimization metrics mentioned above, i.e., $Pr \in \{Time, PDI, Memory, Nodes\}$. Then, the performance of the ML strategy M over the problem p is

$$Pr_M(p) = \begin{cases} Pr_{\mathcal{C}}(p), & \text{if } M(x^p) \leq \tau \\ Pr_{\mathcal{NC}}(p), & \text{if } M(x^p) > \tau, \end{cases}$$

where x^p is the feature vector representing p, while τ is the threshold chosen for discriminating the two methods.
When the model is evaluated in terms of optimization metrics, we average its performance over multiple instances by using the *Shifted Geometric Mean* (*Shm*).

Definition 4.3. *Given a strategy ϕ, a test set of MIP problems $\mathcal{T} \subset \mathcal{P}$ and a performance metric $Pr \in \{Time, PDI, Nodes, Memory\}$, we define the Shifted Geometric Mean of ϕ over $\mathcal{T}$, with metric Pr, as*

$$Shm_{Pr}(\phi) = \left(\prod_{p \in \mathcal{T}} \left(Pr_\phi(p) + S \right) \right)^{\frac{1}{|\mathcal{T}|}} - S,$$

while S is a hyperparameter called shift, whose value is set according to the chosen metric Pr.

Within the optimization community, in particular, the *Shm* is a widely used evaluation method to assess the performance of any optimization solver over a collection of problems. The reason that motivates the use of the *Shm* consists in its insensitivity to both very large and very small outliers, which turns out to be particularly advantageous when averaging widely spread-out values, as the performance measurements of an optimization solver over a given test set usually are.

We conclude this section by observing that all the metrics defined above will be used, in various places, while reporting the results of our computational experiments. However, since the central goal of this thesis is to improve the running time of the solver, we will evaluate our strategies mostly in terms of running time over the individual instances, and in terms of shifted geometric mean of the running times over multiple instances. Hence, we consider this evaluation method as our default one.

Chapter 5

Computational Experiments

In this chapter, we present the most significant experiments conducted during our computational study. In particular, in Section 5.1, we specify the computational environment in which we run our data collection process (see Section 4.3), hence we describe the preprocessing steps that we execute to obtain a suitable dataset for our experiments. Then, we start the presentation of our study by discussing, in Section 5.2, the quality of the learned models in terms of several heterogeneous metrics, both from a machine learning and from an optimization perspective, hence we illustrate a first, general and comprehensive evaluation of our approach. In Section 5.3, instead, we focus on the runtime of the solver, by providing a closer assessment of the impact that our decision has on the solving process. In Section 5.4, we describe and evaluate the most significant one among the alternative methodologies that we attempted, during the preliminary phases of our research, to formulate and solve our decision problem, namely, classification. In particular, we show of the superiority of the regression formulation over the classification one, hence we justify our choice of using the former as our default approach. In Section 5.5, instead, we describe our feature selection process, in which we investigate the predicting power of our features with respect to the output variable. Finally, in Section 5.6, we test the robustness of our method with respect to the underlying solver, that is, we repeat the evaluation of our heuristic solution by using different releases of the FICO Xpress software.

5.1 Computational Setup & Data Preprocessing

In Section 4.3, we show how to generate, for a given MIP solver $\mathcal{S}$ and MIP problem set $\mathcal{P}$, a dataset $\mathcal{D}(\mathcal{S}, \mathcal{P})$ of features-label observations $(x_{\mathcal{S}}^{p_s}, y_{\mathcal{S}}^{p_s})$, with $p_s \in \mathcal{P}^*$, while in Section 4.4, we describe how to apply, on this set, our training and testing methods for the development and the evaluation of our heuristics.
In particular, for the offline computational study presented in this chapter, we use two datasets, $\mathcal{D}(\textit{Xpr8.9}, \textit{Miplib17})$ and $\mathcal{D}(\textit{Xpr8.11}, \textit{Miplib17})$, constructed from the Benchmark Set from MIPLIB 2017 [GHG+21], denoted as *Miplib17*, using two different releases of FICO Xpress: our default one Xpress 8.9, denoted as *Xpr8.9*, and the more recent

one XPRESS 8.11, denoted as *Xpr8.11*. During the data collection process, in particular, each run is executed, for both solvers, on a 2x Intel E5-2640 v4 CPU @ 2.4Ghz with 20 cores, equipped with 32 GB RAM and running a Red Hat Linux 9 as operating system, and with a fixed time limit of two hours, $T = 7200s$, after that the run is stopped and the best solution found so far is returned.
Now, before splitting the two sets as described in Section 4.3.1, hence conducting the experiments presented in this chapter, we perform a data preprocessing phase, in order to prepare the two datasets for our computational study. Precisely, our data preprocess consists of the following steps.

Sanity Check. First of all, we check the correctness and consistency of our data, hence we generate, for each of the two datasets, two different versions, according to the two different criteria adopted for designing the features in the groups Variables and Constraints (see Table 4.1). Precisely, in one case, we compute these features by using, in terms of comparable statistics, an *inclusive* criterion: each binary variable is also counted as an integer one, i.e. `Binaries ≤ Integers`, any set partitioning constraint is also counted as a cardinality one, i.e. `SetPartitioning_Const ≤ Cardinality_Const`, any set packing constraint is also counted as a binary packing one, i.e. `SetPacking_Const ≤ BinaryPacking_Const`, and so on and so forth. Vice versa, in the other case we use a *disjunctive* criterion, which instead produces an exclusive relation between comparable classes of statistics. The experiments presented in this chapter are conducted on the inclusive version of our datasets; note that, however, the obtained results are nearly invariant under switches between the two versions, that is, the same conclusions can be drawn if our experiments are conducted by using the other version.
Data Cleaning. In the second phase of our preprocess, we execute a data cleaning procedure, during which we filter out all the observations that do not provide relevant information to our study. Precisely, from each of the two datasets, we exclude the record $\left(x_{\mathcal{S}}^{p_s}, y_{\mathcal{S}}^{p_s}\right)$ of $p_s \in \mathit{Miplib17}^*$ if:

- the solver $\mathcal{S}$, in at least one of the two configurations, $\mathcal{C}$ or $\mathcal{NC}$, runs out of memory while solving p_s;
- the solver $\mathcal{S}$, regardless of the two configurations, $\mathcal{C}$ and $\mathcal{NC}$, solves p_s already at the root node, before entering the search tree;
- the optimal objective value of the first relaxation of p_s, due to numerical inaccuracy, differs between the two runs of $\mathcal{S}$, $\mathcal{C}$ and $\mathcal{NC}$.

Feature Scaling. In Section 4.1, we describe the 32 features used to represent each MIP problem. Before starting our learning experiments, however, we need to re-scale, for each solver $\mathcal{S}$, some of these features, in order to normalize, across each of the two datasets, the magnitudes and ranges of values of all independent variables, hence to make sure that none of them intrinsically dominates over the others. This is a common practice in machine learning and, in our case, it is particularly necessary for the success of the training algorithms of the linear model and the neural network. Precisely, for each

feature

$$f \in \{\texttt{Rows}, \texttt{Columns}, \texttt{PresolRows}, \texttt{PresolColumns}, \texttt{Coefficient\_Oom}, \texttt{RightHandSide\_Oom}, \texttt{Objective\_Oom}\},$$

and for each instance $p_s \in \mathit{Miplib17}^*$, we replace $f_{\mathcal{S}}^{p_s}$, in the feature vector $x_{\mathcal{S}}^{p_s}$, with the transformed value

$$\texttt{Transform}(f_{\mathcal{S}}^{p_s}) = \frac{f_{\mathcal{S}}^{p_s} - \min \mathcal{F}_{\mathcal{S}}}{\max \mathcal{F}_{\mathcal{S}} - \min \mathcal{F}_{\mathcal{S}}},$$

where $\mathcal{F}_{\mathcal{S}} = \{f_{\mathcal{S}}^{q_t} \mid q_t \in \mathit{Miplib17}^*\}$. Note that, in this way, we have all features ranging within the interval $[0, 1]$, except the feature `Symmetries`, which is instead a binary one (see Table 4.1).

Now, *Xpr8.9* is the only solver that we use for the first 4 experiments of this chapter (Sections 5.2 to 5.5), while *Xpr8.9* and *Xpr8.11* are used, together, only in the last experiment (Section 5.6), in which we compare the quality of our approach between the two solvers. Consequently, we denote as $\mathcal{D}$ the dataset on which we conduct the first 4 experiments, and as $\mathcal{D}^{8.9}$ and $\mathcal{D}^{8.11}$ the datasets on which we conduct the last experiment. More precisely, $\mathcal{D}$ consists of the preprocessed observations of $\mathcal{D}(\mathit{Xpr8.9}, \mathit{Miplib17})$ corresponding to the instances that survived solely the data cleaning process of *Xpr8.9*, while $\mathcal{D}^{8.9}$ and $\mathcal{D}^{8.11}$ consist of the preprocessed observations of $\mathcal{D}(\mathit{Xpr8.9}, \mathit{Miplib17})$ and $\mathcal{D}(\mathit{Xpr8.11}, \mathit{Miplib17})$, respectively, that survived the data cleaning processes of both *Xpr8.9* and *Xpr8.11*. In this experiment indeed, in order to obtain correct and non-misleading results, we need to make sure that the ground set of instances is consistent between the two datasets. Precisely, the instances in the test bed of $\mathcal{D}$ and the ones in the test bed of both $\mathcal{D}^{8.9}$ and $\mathcal{D}^{8.11}$ are listed in Table A.1 and Table A.2 in the appendix, respectively.

To conclude this section, we list few further details to complete the introduction of the computational study presented in this chapter:

- in order to mitigate the impact of the randomness affecting the algorithms that we use for training `RF` and `NN` (the training process of `LM` is deterministic) and to produce more accurate results, we repeat each of the experiments presented in this chapter for 10 random seeds, hence we aggregate the results over the different repetitions;
- as competitors against the learned models, we consider the following two strategies: `Always_Cut`, which chooses the method $\mathcal{C}$ for all input problems, and `Never_Cut`, which, conversely, always chooses the method $\mathcal{NC}$. Moreover, we also take into account the best possible strategy, denoted by `Oracle` and corresponding to the one that always selects the optimal method between $\mathcal{C}$ and $\mathcal{NC}$, hence representing the best that we can achieve.

5.2 Baseline Evaluation

In this section we present some baseline learning experiments, designed to provide a general and extensive evaluation of our approach. More precisely, we train our three models, `LM`, `RF` and `NN`, on the train set $\mathcal{D}_{train}$, hence we evaluate them, by means of different evaluation methods, on both the test set $\mathcal{D}_{test}$, to obtain an unbiased estimation of their generalization capabilities, and on the train set $\mathcal{D}_{train}$, to check any occurrence of overfitting.

First of all, we assess the quality of our approach from a machine learning perspective, hence we evaluate the learned models in terms of a commonly used "machine learning metric", namely, the mean squared error (*MSE*) (Definition 4.2). The results of this evaluation are displayed in the bar plot in Figure 5.1.

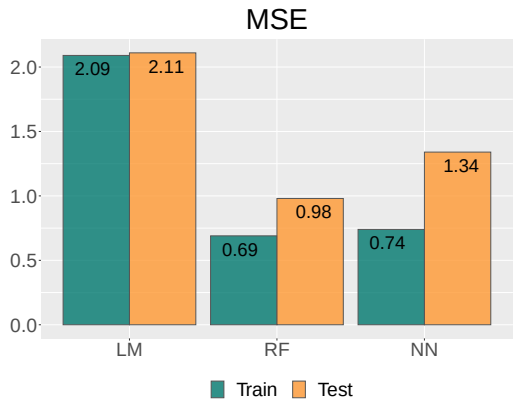

Figure 5.1: the mean squared error of our learners, computed over both the train set $\mathcal{D}_{train}$ and the test set $\mathcal{D}_{test}$.

A first consideration that we can make is that, among the three models, `RF` is clearly the most performant one, being it better than `NN` and `LM` by roughly 7% and 67% on the train set, and by roughly 27% and 53% on the test set, respectively. Moreover, in all models, we can observe a performance degradation from $\mathcal{D}_{train}$ to $\mathcal{D}_{test}$. As we know, however, this is a typical and expected ML phenomenon, and depends on the fact that the instances in D_{test} are completely unknown to the learners, unlike the ones in D_{train} which, instead, have already been seen during the training phase. More precisely, the gap between the train and test error is more evident in the random forest and, especially, in the neural network, for which it can be interpreted as an indication for overfitting. In the linear model, conversely, the rise in the prediction error, from D_{train} to D_{test}, is almost irrelevant; we note however that, for this model, the *MSE* is particularly high, in comparison with the other two models, on both evaluation sets, meaning that, rather than overfitting, `LM` is clearly underfitting the training data, that is, the linear model is neither able to fit the test set nor the train set.

Now, as discussed in Section 4.4.2, other than measuring the average error that our models make when predicting the speedup factor, we are particularly interested in assessing their efficiency when they are directly employed in practical MIP solving, that is, in quantifying the contribution that they provide to the average performance of the solver. To this aim, we now test the learned strategies in terms of some common "optimization metrics", such as the running time (in seconds, up to $T = 7200s$) and the primal-dual integral [Ber13], the number of branch-and-bound nodes and the peak memory usage (in kilobytes), namely, *Time*, *PDI*, *Nodes* and *Memory*, respectively. In Table 5.1 we report the results achieved by our models, our competitors and the best possible strategy, for each metric and on each set. Precisely, in the cases of *Time* and *PDI*, we compute the *Shm* with shift set at 10 over the entire train and test set; in the cases of *Nodes* and *Memory*, instead, we compute the *Shm* with shift set at 1000, and only over those instances that are solved within the time limit by all competing strategies, in order to avoid incorrect and misleading results [1]. Finally, in the last two columns of the table, we report the improvement provided by the best of our models to the performance of the solver (Imp), in comparison with the better of the two competitors, as well as the reduction provided by this model to the performance gap between this competitor and the perfect oracle (Gap). For each row, in particular, the numerical results, shown for the improvement and for the gap reduction, are highlighted in green or red, according to whether the better between our competitors, in red, is beated by the best among our strategies, in green.

By looking at Table 5.1, we can immediately observe that `Always_Cut`, between our two competitors, is certainly the more performant one, being it, on both sets, better than the other by roughly 27% in *Time* and 15% in *PDI*, and by almost 47% in *Nodes* and 7% in *Memory*, hence confirming the results obtained with CPLEX 12.5 by Achterberg and Wunderling ([AW13]), as well as the ones obtained with Xpress 8.9 by the author of this thesis, both discussed in Section 3.1 and both leading to the same conclusion: at least for modern solvers, the local-cut method $\mathcal{C}$, on average, dominates the no-local-cut one $\mathcal{NC}$. In other words, if `Always_Cut` and `Never_Cut` were the only available strategies to take the $\mathcal{C}/\mathcal{NC}$ decision, then we would definitely choose the former. However, by looking at the results scored by our models, and in particular at the ones achieved by `RF` and `NN`, we can immediately realize that a better choice is actually possible.

In terms of *Time*, when compared against `Always_Cut`, the random forest provides a speedup of roughly 8.5% on the train set and of 3.3% on the test set, hence reducing the gap, between the runtime of our main competitor and the best achievable one, by 3/4 and 1/3 on the two sets, respectively. A similar behavior can be observed in `NN`, although this model is slightly less performant than `RF`. In contrast, `LM` can not provide a significant contribution to the running time of the solver; it shows, indeed, results very similar to

[1] Let $n_{\mathscr{A}}$ and $n_{\mathscr{B}}$, with $n_{\mathscr{B}} < n_{\mathscr{A}}$, be the number of nodes explored by solvers $\mathscr{A}$ and $\mathscr{B}$ while running over a certain problem P, respectively. Now, suppose that, on this problem, $\mathscr{A}$ could achieve optimality while $\mathscr{B}$ ran out of time; then, in this case, comparing the two solvers over P, in terms of *Nodes*, would be misleading, since $\mathscr{B}$ would be declared the winner, even though $\mathscr{A}$, over P, actually performed better.

Set	Metric	LM	RF	NN	Always_Cut	Never_Cut	Oracle	Imp (%)	Gap (%)
~ Train Set ~	*Time*	464.51	**427.91**	433.38	**467.92**	642.56	414.75	**-8.55**	**-75.25**
	PDI	84.67	**79.50**	80.87	**84.39**	99.62	76.21	**-5.79**	**-59.78**
	Nodes	23150	**22363**	23148	**22694**	43554	20363	**-1.46**	**-14.2**
	Memory	1131764	1103129	**1092546**	**1152738**	1246085	1008815	**-5.22**	**-41.82**
~ Test Set ~	*Time*	442.42	**420.09**	420.83	**434.61**	593.21	384.72	**-3.34**	**-29.10**
	PDI	81.44	**78.13**	78.72	**79.26**	93.66	72.03	**-1.43**	**-15.63**
	Nodes	**20685**	21243	21303	**19903**	37813	17920	**+3.93**	**+39.44**
	Memory	1081879	**1037462**	1057593	**1086532**	1157677	943191	**-4.52**	**-34.23**

Table 5.1: the performance of the competing strategies, measured in terms of different optimization metrics and aggregated by means of the *Shm* over both sets, together with the contribution provided to the solver by our models.

the `Always_Cut`'s ones, meaning that this model is not really able to distinguish those instances on which the $\mathcal{NC}$ approach might work better than the other, but rather it is very prone to select $\mathcal{C}$ in the majority of cases.
Similar considerations can be made by looking at the *PDI*, even though, in this case, the contribution provided to the solver is less significant than the one observed in *Time*.
In contrast, the results obtained in terms of *Nodes* seem to contradict the ones observed in the two metrics previously discussed. Indeed, with an improvement of less than 1.5% and a gap reduction of less 15%, the *Nodes* performance of our solver is, on $\mathcal{D}_{train}$, almost unaffected by the use of our models, while it is even weakened by them on $\mathcal{D}_{test}$, where we can observe a 4% degradation and a 40% gap increment in the average node consumption. This tendency, however, is explainable by the following fact: the use of cutting planes, as discussed in Section 3.1, has the main advantage of improving the dual bound, hence to increase the chances of pruning, consequently to reduce the number of explored nodes. This is why, when measured in terms of *Nodes*, the `Always_Cut` strategy, which always chooses, by definition, the local-cut approach $\mathcal{C}$, tends to dominate its competitors. Moreover, this explains also why, in terms of *Nodes*, the weakest of our models, `LM`, is very close to the other two on the train set, while it is even better than them on the test set. As already mentioned, indeed, the linear model tends to imitate `Always_Cut`, hence, as a consequence, it shows a more competitive performance when evaluated in terms of *Nodes*.
Nevertheless, the degradation in the average *Nodes* performance of the solver, caused by the introduction of our models, is not reflected in the average amount of memory consumed during the solve. If we compare the competing strategies in terms of *Memory*, indeed, we can confirm the trend that emerged for *Time* and *PDI*, that is, the learned policies evidently contribute to the quality of the solver, with `RF`, in particular, providing an improvement of around 5% and a gap reduction of more than $1/3$ on both sets.

A last, yet very interesting consideration, can be made by looking at the change in the performance of the learned models from the train set to the test set. Indeed, we can clearly observe that, if our strategies are tested directly in the context of practical MIP solving, they score better results on $\mathcal{D}_{test}$ rather than on $\mathcal{D}_{train}$, in evident contrast to what shown by the *MSE* evaluation and, more in general, to what is the typical tendency in machine learning. The contradiction here, however, is only apparent, and can be resolved by the following argumentation: our solver tends to lose performance from $\mathcal{D}_{test}$ to $\mathcal{D}_{train}$, given that, according to the adopted split criterion (Section 4.3.1), the former only contains the original instances of our dataset $\mathcal{D}$ (seed 0), while the latter consists of the remaining randomly permuted ones (seeds 1 to 5). In short, $\mathcal{D}_{test}$ is easier to solve, if compared with $\mathcal{D}_{train}$. This behavior, in particular, mainly depends on two reasons: the use, in XPRESS, of structure-detection algorithms that assume a "natural" input sequence of the problem, and the increase, during the solving procedure, in the number of cache-misses when the problem is permuted. In other words, the observed phenomenon does not have any machine learning nature, but rather an optimization and, more in general, computational one. This is confirmed by the results scored by our competitors and by the best possible strategy. As we can see, indeed, the advance in the solver performance, from $\mathcal{D}_{train}$ to $\mathcal{D}_{test}$, can be observed also in `Always_Cut` and `Never_Cut`, as well as in `Oracle`, even though these strategies are not machine-learned, but rather human-designed. If we ignore the absolute performances, and we look only at the improvement and the gap reduction, we can again recognize the usual, expected behavior of any machine learning application. Indeed, the contribution provided to the solver evidently decreases from $\mathcal{D}_{train}$ to $\mathcal{D}_{test}$, that is, our models tend to lose efficiency on previously unobserved instances.

Now, to provide a more detailed prospect of the contribution given by our approach, we compare the most competitive among our models, `RF`, against our main competitor, `Always_Cut`, on the entire dataset $\mathcal{D}$ and on different *brackets* of this set, as well as on the subset of the *affected* instances. Precisely, for two given solvers, a bracket of a set of problems is defined as follows.

Definition 5.1. *Given two solvers $\mathscr{A}$ and $\mathscr{B}$, a set of problems $\mathcal{T}$ and a fixed time limit T, the bracket $[t_1, t_2]$ is defined as the subset of $\mathcal{T}$ consisting of the instances satisfying the followings:*

- *they are solved (within the time limit T) by either $\mathscr{A}$ or $\mathscr{B}$;*
- *the runtime of the slower between the two solvers is, on them, between t_1 and t_2.*

for $t_1 \leq t_2 \leq T$.

Usually, the right-hand side of each bracket is fixed at the time limit, i.e., $t_2 = T$, while the left-hand one is increased progressively, so to define a hierarchy of subsets of $\mathcal{T}$ of increasing difficulty. In our case, we consider the brackets of our dataset $\mathcal{D}$ obtained by fixing t_2 at our time limit 7200, hence by assigning to t_1 the following

values: $0, 1, 10, 100, 1000, 2000$. The subset of the affected instances, instead, consists of the ones that are solved by at least one of the two competitors, and for which these competitors make opposite choices, i.e., the ones on which `RF` decides to not cut. In other words, these are the instances on which the behavior of the solver, working with the $\mathcal{C}$ configuration by default (`Always_Cut`), is directly modified by the use of the learned heuristic (`RF`).
On each of the considered sets, we compute, for both `RF` and `Always_Cut`, the number of solved instances and the *Shm* of the running times, hence the improvement provided to the solver by the former, when compared against the latter. The results of this comparison are reported in Table 5.2, where "All" and "Affected" denote the entire $\mathcal{D}$ and the subset of affected instances, respectively. Note that "Affected" is, by definition, a subset of $[0, 7200]$, and that the difference between "All" and $[0, 7200]$ is that the former includes all the available instances, while the latter includes only those instances which are solved by at least one of the two strategies. In particular, the number of instances that, out of the entire $\mathcal{D}$, are solved (within the time limit) by both `RF` and `Always_Cut`, by none of them, and by one of them but not the other, are 852, 281, 22, respectively; these numbers, of course, sum up to $1155 = |\mathcal{D}|$ (see Table A.1).

		`RF`		`Always_Cut`		
Bracket	**Instances**	**Solved**	**Time**	**Solved**	**Time**	**Imp.** (%)
All	1155	872	426.59	854	462.16	-7.70
$[0, 7200]$	874	872	167.22	854	186.55	-10.36
$[1, 7200]$	865	863	172.53	845	192.65	-10.44
$[10, 7200]$	753	751	257.15	733	290.20	-11.39
$[100, 7200]$	470	468	750.73	450	874.37	-14.14
$[1000, 7200]$	226	224	2198.42	206	2703.86	-18.69
$[2000, 7200]$	150	148	2893.07	130	3805.64	-23.98
Affected	310	308	132.45	290	180.63	-26.67

Table 5.2: comparison between `RF` and `Always_Cut` , on different brackets of $\mathcal{D}$.

As we can see from Table 5.2, `RF` is able to solve 18 instances more than `Always_Cut`. Moreover, on the entire dataset $\mathcal{D}$, `RF` provides to the average runtime of the solver a 7.7% speedup, which, however, increases to more than 10% on the instances that are solved by at least one of the two competitors ($[0, 7200]$), showing that the number obtained on the whole $\mathcal{D}$ is dampened by the fact that almost 25% of the instances of this set can not be solved, within the time limit, by any of the two strategies. The percentage of improvement keeps increasing together with the hardness of the evaluation set, until reaching an encouraging value of 24% on the most difficult among our problems. Finally,

by restricting the test bed of the comparison to the affected instances, that is, the ones for which `RF` chooses $\mathcal{NC}$ rather than $\mathcal{C}$, hence the ones on which we are, in effect, modifying the solver's behavior, we can observe a very promising improvement of more than 26%.

Conclusions

The results shown in this section suggest the following encouraging conclusions. While the linear model is not really able to discriminate, instance by instance, between the two methods $\mathcal{C}$ and $\mathcal{NC}$, evidently due to the fact that our decision problem is too complex to be solved by a linear regression, the neural network, and in particular the random forest, prove to be valid policies for the algorithmic decision that we want to take. The random forest, in particular in terms of running time, is effectively able to provide significant benefits to the average performance of FICO Xpress, especially over those instances that are particularly hard for the solver.

5.3 Evaluation With Different Thresholds

This section is meant to provide a closer and visual assessment of the impact that our learned solution has on the performance of the solver, in particular on its average running time, whose speedup represents, as we know, the primary objective of our research. Precisely, the experiment that we propose consists in manipulating a crucial hyperparameter of our decision process, namely, the *threshold* used to mark the separation between the two choices, local-cut and no-local-cut, hence in observing the changes in the performance of the solver, in order to realize how influential our decision is able to be.

As described in Chapter 4, the method selection problem has been modeled as a regression problem, where the ML models predict the speedup factors between the runtimes of the two methods $\mathcal{C}$ and $\mathcal{NC}$, instead of predicting directly one of them. Hence, to take the final choice, we fix a threshold for the predicted speedup, representing the switch between the two methods. In other words, this threshold is what we use to turn the regression into a classification.
Precisely, given a feature vector x, the model M predicts the speedup factor $M(x)$, which is then compared with the threshold τ; hence, according to this comparison, the final choice for x is taken (see Figure 4.4).
The value of τ indicates how much we are inclined to discriminate one method in favor of the other. By definition (see 4.1), the speedup factor is negative when $\mathcal{C}$ is faster than $\mathcal{NC}$, positive otherwise; if it is zero, or near zero, then the two methods are equivalent, or roughly equivalent. Hence, the tuned value for the threshold is $\tau = 0$; by untuning it, instead, we introduce some skew in our decision.
By observing the change in the performance of the learned strategies, produced by moving the threshold towards the negatives or the positives, we can evaluate the impact of

our decision on the efficiency of the solver.

In this experiment, we run Algorithm 4 twice for each of our three models, `LM`, `RF` and `NN`. In particular, the models are trained on the train set $\mathcal{D}_{train}$ and evaluated, in the terms of runtime, on the entire dataset $\mathcal{D}$. Finally, the array chosen for the thresholds is

$$V = \big[i/10, \ \text{for integer}\, i \in [-40, 40]\big],$$

that is, the sequence of values between -4 and 4 with a step of 0.1. Note that, although the speedup factors over $\mathcal{D}$ span across a much larger range, the interval $[-4, 4]$ contains more than 90% of them.

Algorithm 4: Evaluation With Different Thresholds

Input: ML model M,
array of threshold values V,
sets $\mathcal{D}_{train}$, $\mathcal{D}$

Output: array $S : S[i] = (\mathit{Shm}$ of M over $\mathcal{D}$ with threshold $V[i])$

Initialization: $T_{\mathcal{C}} \leftarrow [\text{runtime of } \mathcal{C} \text{ over } x, \text{ for } x \in \mathcal{D}]$,
$T_{\mathcal{NC}} \leftarrow [\text{runtime of } \mathcal{NC} \text{ over } x, \text{ for } x \in \mathcal{D}]$,
$P, S \leftarrow [\ ], [\ ]$

1. Train M on $\mathcal{D}_{train}$ [*Training*]

2. For $x \in \mathcal{D}$: $P[x] \leftarrow$ predicted speedup $M(x)$ [*Predicting*]

3. For $i \leftarrow 1$ to $len(V)$ [*Threshold Tuning*]

3.1. $T_M \leftarrow [\ ]$

3.2. For $x \in \mathcal{D}$ [*Classification*]

If $P[x] \leq V[i]$: $T_M[x] \leftarrow T_{\mathcal{C}}[x]$

Else: $T_M[x] \leftarrow T_{\mathcal{NC}}[x]$

3.3. $S[i] \leftarrow \mathit{Shm}$ over T_M [*Evaluation*]

4. Return S

Figure 5.2 shows the results of the experiment. The y-axis represents the output S (Shm of Time), while the x-axis refers to the input V (threshold). The continuous curves are the Shm of the learned strategies; precisely, the three tones of green, from the darkest to the lightest, refer to `RF`, `NN` and `LM`, respectively. The dashed lines in the three shades of blue, again from the darkest to the lightest, are instead the perfect `Oracle` and our two competitors, `Always_Cut` and `Never_Cut` .

We can immediately observe how the Shm of our models converge to `Never_Cut`, on the left, and to `Always_Cut`, on the right. As already mentioned, by moving the threshold to the negatives, we bias the decision towards the method $\mathcal{NC}$. For example, if

the threshold is fixed at $\tau = -2$, then we choose $\mathcal{NC}$ not only when the predicted speedup is larger than 0, but also when it falls between -2 and 0, even though, in this case, we should have chosen $\mathcal{C}$. The smaller the value of τ, the more prone we are to choose $\mathcal{NC}$, that is, we tend to imitate the behavior of `Never_Cut`, and this explains why the curves convergence to this strategy on the negative side. A similar argument holds for the other side, where the curves converge to `Always_Cut`.

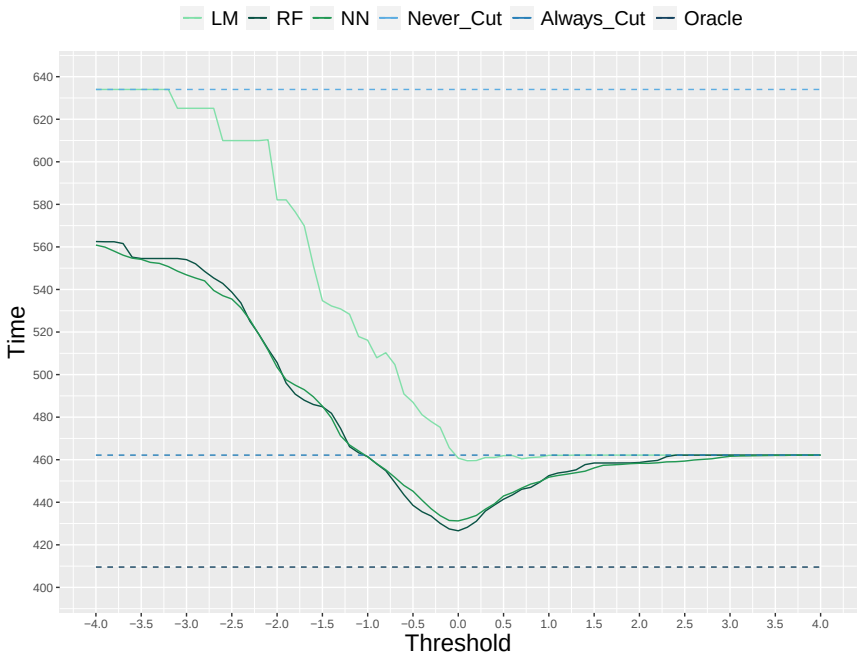

Figure 5.2: threshold tuning over $\mathcal{D}$

It is by looking at the behavior of the curves around zero, however, that we can grasp how influential our decision can be on the performance of FICO Xpress.
In particular, the linear model remains quite close to `Always_Cut`, hence confirming the trend observed in Section 5.2, that is, `LM` is not really able to distinguish the $\mathcal{C}$ instances from the $\mathcal{NC}$ one, but it can simply recognize that the former works better in the majority of cases, hence it is prone to always chose this one. The other two curves, instead, reach their minimal values below the line of `Always_Cut`, showing that `RF` and `NN` are effectively able to contribute to the solver and to reduce the gap between the best of our competitors and the best that we can achieve. More precisely, `RF` provides a 7.7% speedup and a 67.6% gap reduction to the runtime of the solver.

Conclusions

The bar plot displayed above suggests the following encouraging conclusion: if we had to choose between always cutting and never cutting, then we would certainly choose the former, being it evidently the better of the two solutions; an even better one, however, is

to decide, instance by instance, whether to cut or not by means of our learned policies, that is, to let the neural network and, especially, the random forest make their choices, since they have shown, on the considered dataset, to be definitely able to speedup the running time of the solver.

5.4 Classification Experiments

During the preliminary phase of our project, we implemented and tested different approaches for the algorithmic decision that we want to take. In particular, among the various alternatives that we considered, the problem formulation described in Chapter 4 proved to be the most successful one, this is why, in the end, we adopted this as our default methodology. Not every one of the rejected approaches, however, could be asserted as completely ineffective, with few of them being indeed able to achieve acceptable results. In this section, we describe one of these alternative methodologies that we believe to be particularly worth discussing, i.e., the classification approach, hence we test and compare it against the regression one, whose evaluation is illustrated in Section 5.2, in order to show the superiority of our default methodology, hence to motivate our choice.

Undoubtedly, the most natural way to approach our problem is to formulate it as a classification learning task, given that, as described in Section 3.1, the problem itself consists, precisely, in taking a binary algorithmic decision. Indeed, instead of learning to predict the speedup factor between the runtimes of the local-cut and the no-local-cut method, then converting the learned regressor into a binary classifier by means of our a threshold t, that works as a switch between the two methods, we could learn to directly predict one of the two classes, $\mathcal{C}$ and $\mathcal{NC}$.
Precisely, for this experiment, we substitute, in each observation of our default dataset $\mathcal{D}$, the speedup factor defined in (4.1) with either one of the two classes -1 and 1, with the former representing $\mathcal{C}$ while the latter $\mathcal{NC}$, according to which of the two methods is the faster one on the corresponding problem. More formally, we define the class of a given problem p as

$$c_{\mathcal{S}}^{p} = \begin{cases} -1 & \text{if} \quad Time_{\mathcal{C}}(p) \leq Time_{\mathcal{NC}}(p), \\ 1 & \text{otherwise.} \end{cases}$$

Then, on $\mathcal{D}_{train}$, we train again the three machine learning models, linear model (`LM`), random forest (`RF`) and neural network (`NN`), this time, however, in their classification version. Precisely, in the `R` programming language, we perform the trainings of `LM` and `RF` by means of the `train` function provided by the `caret` package [Kuh20], with method `"lda"` ("linear discriminant analysis") and `"rf"`, respectively, while for `NN`, we use the `neuralnet` function provided by the `neuralnet` package [FGW19], similarly to what described in Section 4.4.1.
As evaluation method, we compute the performance of a learned classifier M over a

problem p, in terms of runtime, as

$$Time_M(p) = \begin{cases} Time_{\mathcal{C}}(p) & \text{if} \quad M(x_{\mathcal{S}}^p) = -1, \\ Time_{\mathcal{NC}}(p) & \text{if} \quad M(x_{\mathcal{S}}^p) = 1, \end{cases}$$

where $x_{\mathcal{S}}^p$ is the feature vector representing p. Finally, we aggregate the results, obtained over a set of multiple instances, by using the shifted geometric mean (*Shm*), with shift set at 10. As explained in Section 5.1, in this experiment we have $\mathcal{S}$ = *Xpr8.9*.

In Table 5.3 we report, on both $\mathcal{D}_{train}$ and $\mathcal{D}_{test}$, the results achieved by the three classifiers and the ones, already discussed in Section 5.2, scored by the three regressors and their main competitor, as well as by the best possible strategy. Moreover, to compare the classification approach against the regression one on both sets, we highlight, in green, both the numerical result corresponding to the best among the six learned strategies, i.e., the three regressors and the three classifiers, and the improvement and the gap reduction, reported in the last two columns, provided by this model.

Set	**Version**	LM	RF	NN	Always_Cut	Oracle	**Imp** (%)	**Gap** (%)
~ *Train Set* ~	Regression	464.51	**427.91**	433.38	467.92	414.75	**-8.55**	**-75.25**
	Classification	462.22	435.00	439.61			-7.04	-61.91
~ *Test Set* ~	Regression	442.42	**420.09**	420.83	434.61	384.72	**-3.34**	**-29.10**
	Classification	428.23	432.54	431.81			-1.47	-12.79

Table 5.3: comparison between regressors and classifiers, in terms of the *Shm* of the runtime over $\mathcal{D}$.

As clearly shown by the table, the classification approach is able to guarantee positive results, and to represent an acceptable solution method for our decision problem. The three classifiers, indeed, are all able to perform better than their competitor on both sets and to guarantee some improvement to the average runtime of the solver. It is evident, however, that the classification approach is not able to compete with the regression one. The random forest and the neural network clearly suffer from the alternative formulation of our problem. Similarly to each other, indeed, they lose performance by roughly 1.5% on the train set, while on the test set, the percentage of performance deterioration rises to the disappointing value of 3%. The only one, among the learned models, that seems to benefit from the classification formulation is the linear model, with the linear classifier showing, in comparison with its regression counterpart, stronger generalization

capabilities. This observation, however, is not really significant, since the linear model shows an unsatisfactory performance already in regression.
By comparing the six learners, i.e., the three regressors and the three classifiers, among each other, we can realize that, on both sets, the random forest regressor is clearly the best of them, immediately followed by the neural network regressor. These two regressors represent, indeed, the most competitive and promising heuristics produced by our approach, and provide evidence of the superiority of the regression method over the classification regressor.

Now, two main argumentations explain why the regression approach actually outperforms the classification one, even though the latter should be, intuitively, more suitable for the decision problem that we want to solve, given the nature of the problem itself.
First of all, since the central goal of our study is to speed up the solving process of XPRESS, we are evaluating our approach in terms of runtime, that is, we are comparing the three regressors and the three classifiers, against each other, in terms of a regression metric, for which the regressors naturally tend to work better. If we repeat the comparison by using, this time, any classification metric, we can clearly observe the opposite trend. This is confirmed by Figure 5.3, where we report the performance of our models, over the entire dataset $\mathcal{D}$, in terms of *accuracy*, defined as the number of correctly classified instances over the total number of instances.

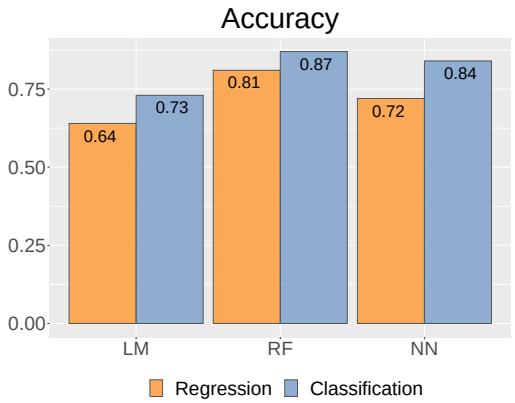

Figure 5.3: when compared in terms of a classification metric (over the entire dataset $\mathcal{D}$), each classification model outperforms its regression counterpart.

Secondly, the quality of our models mostly depends on how they behave on those instances on which the performances of the two methods significantly diverge from each other, rather than on the ones on which the two methods perform similarly, i.e, the borderline instances. That is, a wrong prediction on an instance with a high speedup factor impacts the performance of our model much more negatively than a wrong prediction on a borderline instance. Now, the regression training, unlike the classification one, is driven especially by the relevant cases, rather than by the ignorable ones, that is, the regressors focus their learning power on those cases on which a good performance is decisive. This

was confirmed by a minor experiment that we conducted in our study (not reported in this thesis), in which we performed a misclassification analysis of our models, both in regression and classification. From the results of the experiment, we could observe that the regressors tend to fail especially on those instances with a speedup very close to 0, while for the latter we could observe exactly the opposite situation. This is why the regression models, in terms of runtime, tend to outperform their classification counterparts.

Conclusions

The classification methodology proves to be a valid solution approach for our decision problem, being it able to produce competitive models that can improve, or at least not damage, the average performance of the solver. This approach, however, is evidently not able to outperform the regression one, given that, among the six learned strategies, the most competitive ones remain the regression version of the random forest and the neural network. This, in particular, motivates our preference for the regression formulation over the classification one, hence justifies our choice of selecting the former as our default methodology.

Now, as a consequence of the results observed from the first three experiments presented in this chapter, we can definitely conclude, at the end of this section, that the linear model can not really provide any contribution to the solver, being our decision problem, evidently, too complex to be solved by a linear approach. The neural network, instead, represents a valid solution to the problem, being it able to produce valid policies that can effectively contribute to the improvement of the solver. Despite the capabilities demonstrated by this model, however, it is undoubted that the random forest is, especially in regression, the most competitive one among the three learners, as shown by all evaluations presented so far.
To simplify the discussion, in the remaining of this chapter we will focus solely on the random forest, unless explicitly mentioned otherwise.

5.5 Feature Selection

Feature selection is the process of identifying and selecting, among the originally considered features, the most relevant and consistent ones for the variable(s) to predict. The feature selection procedure represents a fundamental building block of modern machine learning workflows, and it is motivated by several reasons, such as:

- simplifying the adopted model and improving its interpretability, other than decreasing the training time;
- reducing overfitting, hence enhancing the generalization capabilities of the learner;
- preventing the curse of dimensionality [Bel61];

- reducing the computational cost of the feature extraction process for future real-world deployments of the produced model;
- more in general, refining the overall understanding of the problem, by recognizing those pieces of information which the response variable(s) especially depends on.

For us, the last two of the motivations listed above are particularly relevant, that is, by including a feature selection step into the development pipeline of our ML solution, we aim at a both practical and theoretical goal. On one side, indeed, we want to reduce the number of attributes used to describe our input instances, in order to relieve the computational effort necessary to extract them, given that the ultimate goal of the present work is to integrate the final predictive tool into the FICO Xpress solver, where the extraction process needs to be executed online, i.e., during the actual MIP solve. On the other side, we want to achieve a stronger comprehension of our problem, as well as of the relations among the pieces of information that we have available.

In the experiment proposed in this section, we develop a method to measure, from the model's perspective, how explanatory each input variable is for the output one. Then, we re-train the model several times, each time by using a reduced number of features, selected among the most important ones, hence we re-evaluate the model and we observe how its performance reacts to the feature reduction, in order to understand how hardly our feature set can be reduced, hence to check whether the feature extraction process, in the deployment of our solution, can effectively be simplified.
In particular, after training our random forest on the train set $\mathcal{D}_{train}$ by using the original set of features, we perform a feature ranking process in which we assign a score to each predictor, in order to measure how informative this is for the decision to take or, more briefly, to quantify its predictive power. Precisely, the relevance of an attribute is measured as the mean decrease in impurity [Lou15], that is, in terms of the depths in the trees of the ensemble where the attribute is used for the node splits, with the rationale that the more superficially an attribute is used in a decision-tree, the larger the number of samples that it affects, hence the more influential it is for the decision to take. More specifically, we perform the ranking procedure by means of the `varImp` function provided by the `caret` package [Kuh20], hence we obtain a scoring of our features consisting of non-negative scalar values that sum up to 1. The results of the feature ranking procedure, for our model `RF`, are displayed in the bar plot in Figure 5.4, where the 32 attributes are listed, top-to-bottom, in decreasing order of importance.

From the bar plot, it seems that our random forest, in taking its decision, is particularly influenced by the basic information on the matrix of the problem, as well as its computational behavior, rather than its combinatorial structure. Among the first positions in our feature ranking, indeed, we find the features describing the size and the sparsity of the matrix, `Rows`, `Columns` and `NonZeros`, the size of its presolved matrix, hence the activity of the presolver, `PresolRows` and `PresolColumns`, as well as the gaps between the different objective values at the end of the root node, hence the impact of

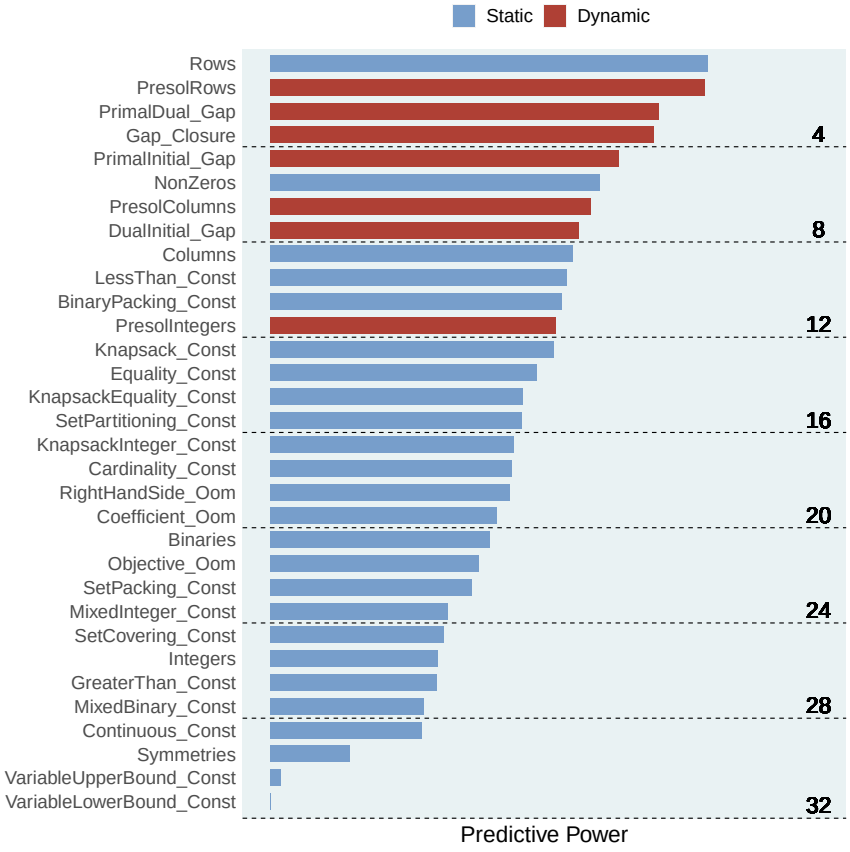

Figure 5.4: importance of our features from the RF perspective.

the global cutting loop, `PrimalInitial_Gap`, `PrimalDual_Gap`, `DualInitial_Gap` and `Gap_Closure`. The dynamic information about the problem, in particular, clearly dominates the static one, that is, the discrimination between the two methods heavily depends on the behavior of the underlying solver on the problem, rather than on the structure of the problem itself.

Now, in the second part of this experiment, we repeat the training/testing process of our model several times, each time by using only the n most important features, for $n = 32, 28, 24, 20, 16, 12, 8, 4$, selected according to the feature importance ranking displayed in Figure 5.4. In other words, we recursively reduce our feature set by 12.5%, hence we observe the impact of the reduction on the quality of our model. Figure 5.5 depicts how the performance of the random forest, measured in terms of the *Shm* of the runtime over our dataset $\mathcal{D}$, changes after each iteration of our feature reduction process.

As clearly shown by the plot, the RF performance slightly improves after the reduction of the features. Precisely, the minimum *Shm* value is reached by the model trained on the 8 most important features; this model, in particular, is roughly 1% faster than the

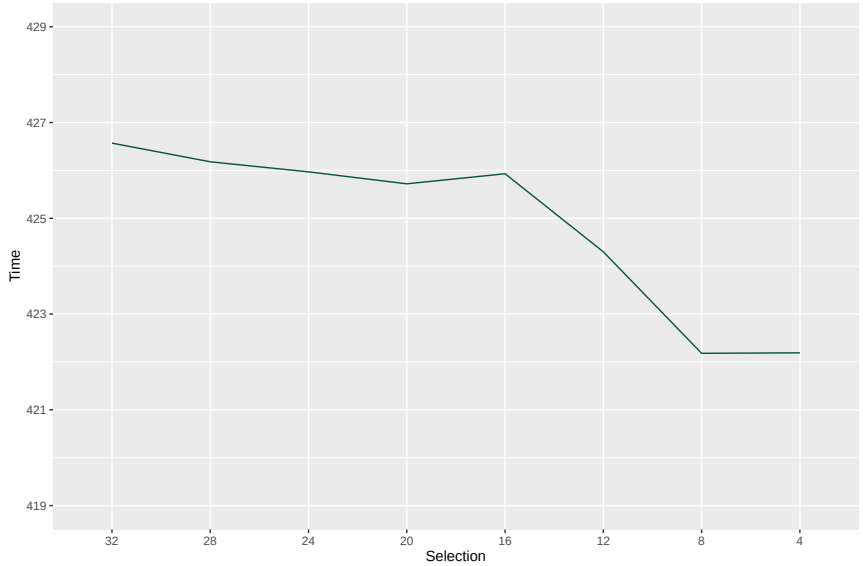

Figure 5.5: The change in the performance of the random forest over $\mathcal{D}$ during our feature reduction process.

one trained on the original feature set. This shows that our random forest is able to preserve its quality even when it is forced to learn over a significantly reduced amount of information. This, in particular, allows for an effective simplification of our problem formulation, as well as for a substantial mitigation of the feature extraction process, that needs to be executed online once that the model is deployed for practical MIP solving.

Conclusions

From the experiment presented in this section, we can draw the following conclusions. On one side, our model seems to react very well to our feature reduction process, hence allowing for a substantial decrease in the amount of information necessary for an effective discrimination between the two methods, which turns out to be particularly convenient in the deployment stage. On the other side, however, in order to correctly make our choice, the computational behavior of the problem can not be ignored, that is, the decision that we are investigating seems to significantly depend on the particular solver in use. A question that naturally arises, at this point, is whether the model, trained for one solver, can be transferred and adapted to different solvers, without losing its competitive performance. This question represents one of the motivations for the experiment presented in the following section, in which we will investigate how the quality of our approach changes from one solver to another.

5.6 Evaluation With Different Solver Releases

The methodological approach described in Chapter 4, hence the solution provided to the decision problem defined in Chapter 3, have been implemented, basically, for FICO Xpress 8.9 (*Xpr8.9*), representing our main technological support. With the research project formalized in this thesis, however, we have the higher ambition of developing, for the mentioned problem, a powerful and more general solving tool that, independently of the particular version of Xpress in use, can be permanently integrated into the solver framework in order to contribute to its improvement. A question that arises naturally, in these terms, is whether the encouraging results, achieved by our approach for Xpress 8.9, can be confirmed also for other, especially more recent, releases of the same software. This question, in particular, is further motivated by the results obtained from the feature selection experiment discussed in Section 5.5, where we observe that our predictive model `RF` significantly depends on those features representing the behavior of the solver on the input instance. More precisely, with the experiment presented in this section, we aim at answering the following questions.

- Can our approach be entirely re-implemented for another solver version, without altering the quality shown for our default version?
- How does the efficacy of our solution, implemented for a certain solver release, change when it is transferred to a different release, that is, when the underlying solver undergoes a significant change between the training and testing phase of our model?

In order to answer these questions, we consider two versions of Xpress, namely, *Xpr8.9*, our default release, and *Xpr8.11*, the second most recent one at the time of writing this thesis. In particular, we train our random forest once on $\mathcal{D}^{8.9}_{train}$ and once on $\mathcal{D}^{8.11}_{train}$, hence we evaluate each of the two produced strategies on both $\mathcal{D}^{8.9}$ and $\mathcal{D}^{8.11}$, where $\mathcal{D}^{8.9}$ and $\mathcal{D}^{8.11}$ are, as described in Section 5.1, the datasets collected by running, on our ground problem set, the two solvers *Xpr8.9* and *Xpr8.11*, respectively.
In other words, the experiment consists in training/evaluating our random forest, once by keeping the solver unchanged, and once, instead, by switching it between the two phases. More precisely, by training and testing our model with the same solver, we aim at discovering whether our solution can be entirely re-implemented in a different solver release, without observing any loss in its quality, hence at answering the first of the two questions listed above. By changing the solver between the training and the testing phase, instead, the goal is to find out whether the trained heuristic can be eventually transferred, without negative consequences, from one solver release to another or, equivalently, to test its robustness with respect to the solver, hence to answer the second question. The results of the experiment are reported in Table 5.4.

As shown by the upper side of the table, our approach proves to be a valid one even for other versions of Xpress. The random forest, indeed, is able to guarantee, to the more recent release of the software, the same positive contribution that provides to the older one. The same considerations, however, can not be made when the predictive tool is

Train/Test	RF	Always_Cut	Oracle	Imp (%)	Gap (%)
8.9/8.9	399.45	435.39	384.45	-8.26	-70.56
8.11/8.11	375.74	407.87	364.05	-7.88	-73.32
8.9/8.11	401.05	407.87	364.05	-1.67	-15.58
8.11/8.9	427.00	435.39	384.45	-1.93	-16.48

Table 5.4: The performance of the random forest when it is trained and tested by using the same solver, as well as when it is transferred from one solver to another.

transferred from one solver to another, that is, when it is designed for one solver version, but then employed for a different one. As shown by the lower side of the table, indeed, when trained for Xpress 8.9 but tested for Xpress 8.11, as well as in the opposite situation, the improvement and gap reduction provided to the solver by our random forest roughly deteriorate from 8% to 1.5%, and from 70% to 15%, respectively.
One possible explanation of the observed phenomenon is that the decision that we want to take simply can not disregard the solver in use. In particular, this is consistent with the results observed in Section 5.5, that is, the information about the mathematical formulation of the input problem is not enough to effectively discriminate between the two methods $\mathcal{C}$ and $\mathcal{NC}$. In other words, the decision that we want to take heavily depends on the underlying solver, hence the model loses performance when this solver is substituted with a different one. A second interpretation is that the optimistic results reported in the upper side of the table are slightly heightened by the occurrence of overfitting; in this side indeed, unlike in the lower one, the model is evaluated on a relevant percentage of instances that have already been seen during its training. Despite this quality deterioration, however, we observe that the learned heuristic is still able to improve the solver upon the existing strategies.

Conclusions

The experiment presented in this section demonstrates that the quality of our solution can be replicated also for another, more recent release of FICO Xpress. When, instead, our model is transferred between different solver versions, it suffers an evident quality deterioration. Even in this case, however, the model is still able to provide the solver with a positive contribution. In other words, our random forest has the potential to be generalized across different solver releases, meaning that it can be implemented and integrated into the software permanently, that is, it does not need to be re-trained in each development cycle. This represents, undoubtedly, a very promising conclusion of this experiment.

Conclusions and Outlook

In this thesis we provided a data-driven answer to a critical algorithmic question that arises during the solving process of a mixed-integer linear programming problem, namely, whether to use a branch-and-cut or rather a cut-and-branch algorithm. To the best of the author's knowledge, the MIP community is still suffering the lack of a satisfactory comprehension of this specific question for general MIP problems.

In particular, we stated and discussed the decision problem representing the object of the present study, *local-cut vs no-local-cut*, and we showed that the policy consisting in always choosing the local-cut method (always cutting), although outperforming, on average, its counterpart (never cutting), is far from being the optimal strategy for our decision. Hence, we learned to discriminate between the two methods by developing, within a supervised regression framework, three machine learning models, Linear Model, Random Forest and Neural Network.

The results obtained from our computational study, conducted on a large test bed of MIP problems with Xpress 8.9, led to the following conclusions. The linear model is not really able to discriminate between the two approaches, local-cut and no-local-cut, evidently due to the fact that our decision problem is too complex to be solved by a linear regression. The neural network and especially the random forest, instead, represent valid policies for the decision that we want to take. When compared against our main competitor, i.e., always cutting, the random forest is able to provide to the runtime of the solver, averaged over the considered dataset, a 7.7% speedup. This improvement, in particular, increases to the encouraging value of 24% when the evaluation is restricted to those instances that are particularly hard for the solver. The random forest, moreover, guarantees a 68% reduction in the solver performance gap between the always cutting strategy and the best possible one, that is, the "perfect oracle" that always makes the optimal choice between the two methods, local-cut and no-local-cut, hence representing the best that we can achieve. In other words, if always cutting and never cutting were the only available strategies for our decision, then we would certainly choose the former, being it the better solution between the two. An even better solution, however, is to make our random forest decide, instance by instance, when to use local cuts and when, instead, to refrain from it. Finally, the random forest is able to preserve its competitive performance even when it is trained over a reduced number of features, hence allowing

for an effective simplification of our problem formulation, as well as when it is designed and evaluated for Xpress 8.11, hence showing that the quality of our solution can be replicated also for another, more recent release of the same software. When, instead, the underlying solver undergoes a significant change between the training and testing phase, that is, when the trained model is transferred from one solver version to another, it evidently suffers a quality deterioration. However, even in this case, our model is still able to improve the solver performance upon the existing strategies.

The promising results achieved in our study demonstrate the possibility to learn intelligent algorithms for taking crucial decisions that arise from practical MIP solving. In fact, a variant of the random forest suggested in this thesis has already been implemented by the development team of FICO Xpress, and will represent one of the main features to be released with the next version of the solver. There are, however, many potential sources of improvement, in contemporary optimization frameworks, that a machine learning approach particularly suits, hence many fruitful directions in which the interplay between these two disciplines might evolve.
For example, one direct extension of the present work consists in learning to choose, for the input problem, the maximum depth of the B&B search tree in which the cutting procedure should be stopped, rather than choosing only whether to deactivate the procedure after the root node or not, hence in generalizing our binary decision to an integer one. More in general, the methodology proposed in our study can be potentially extended to all those decision problems that fall into the context of algorithm configuration (or solver parameter tuning), such as the problem of deciding which presolving techniques to apply or the one of configuring the details of the underlying linear solver. In these problems, machine learning can help in predicting, before starting the run or at some specific point during the run itself (as in our case), the most performant parameter configuration for the given MIP instance, hence providing the solver with greater flexibility and adaptivity.

Beyond the specific lines of research encouraged by our study, we strongly believe that the technology resulting from the hybridization of machine learning and combinatorial optimization will be playing an ever more leading role in pushing forward the state of the art of Artificial Intelligence.

Appendix A

Ground Problem Sets

In this chapter, we report the composition of the ground problem set of $\mathcal{D}$, used for the experiments presented from Section 5.2 to 5.5, and the ground problem set of both $\mathcal{D}^{8.9}$ and $\mathcal{D}^{8.11}$, used instead for the experiment presented in section 5.6. Precisely:

- Table A.1 reports the instances from *Miplib17**, that survived the data cleaning process of solely *Xpr8.9*;
- Table A.2 reports the instances from *Miplib17**, that survived the data cleaning process of both *Xpr8.9* and *Xpr8.11*.

In both tables, the column "Name" refers to the name of the problems from *Miplib17*, while "Freq" specifies the number of permuted instances that *Miplib17** contains for the corresponding problem.

Name	Freq	Name	Freq	Name	Freq	Name	Freq	Name	Freq	Name	Freq
30n20b8	6	eil33-2	6	mas76	6	neos-4387871-tavua	6	peg-solitaire-a3	6	sing326	6
50v-10	6	eilA101-2	6	mcsched	6	neos-4532248-waihi	6	pg	6	sing44	6
academictimetablesmall	6	enlight_hard	6	mik-250-20-75-4	6	neos-4647030-tutaki	6	pg5_34	6	sorrell3	6
app1-2	6	fast0507	6	milo-v12-6-r2-40-1	6	neos-4722843-widden	6	physician…	6	sp97ar_dup1	6
assign1-5-8	6	fastxgemm-n2r6s0t2	6	momentum1	6	neos-4738912-atrato	6	piperout-08	1	sp98ar_dup1	6
atlanta-ip	6	fhnw-binpack4-4	6	mushroom-best	6	neos-4763324-toguru	6	pk1	6	square41	6
b1c1s1	6	fhnw-binpack4-48	6	n2seq36q	2	neos-4954672-berkel	6	proteindesign121hz512p9	6	square47	6
bab2	6	gen-ip002	6	n3div36	6	neos-5052403-cygnet	5	proteindesign122trx11p8	6	supportcase10	6
bab6	6	gen-ip054	6	n5-3	6	neos-5093327-huahum	5	qap10	6	supportcase12	6
binkar10_1	6	germanrr	6	n9-3	6	neos-5107597-kakapo	6	radiationm18-12-05	6	supportcase18	6
blp-ar98	6	glass-sc	6	neos-1122047	6	neos-5188808-nattai	6	radiationm40-10-02	6	supportcase19	1
blp-ic98	6	glass4_dup2	6	neos-1171737	6	neos-5195221-niemur	6	rail01	2	supportcase22	5
bnatt400	6	gmu-35-40	6	neos-1354092	6	neos-662469	6	rail02	6	supportcase26	6
bnatt500	6	gmu-35-50	6	neos-1445765	6	neos-848589	6	rail507	6	supportcase33	6
bppc4-08	6	graph20-20-1rand	6	neos-1456979	6	neos-873061	6	ran14x18-disj-8	6	supportcase40	6
brazil3	6	graphdraw-domain	6	neos-1582420	6	neos-911970	6	rd-rplusc-21	6	supportcase6	6
buildingenergy	6	h80x6320d	6	neos-2657525-crna	6	neos-933966	6	reblock115	6	supportcase7	6
chromaticindex1024-7	6	highschool1-aigio	6	neos-2746589-doon	6	neos-950242	1	rmatr100-p10	6	swath1	6
chromaticindex512-7	6	ic97_potential	6	neos-2978193-inde	6	neos-957323	4	rmatr200-p5	6	swath3	6
cmflsp50-24-8-8	6	icir97_tension_dup1	6	neos-3004026-krka	5	neos-960392	1	rocI-4-11	6	tbfp-network	6
cms750_4	6	irish-electricity	6	neos-3024952-loue	6	neos17	6	rocII-5-11	6	thor50dday	6
co-100	6	irp	1	neos-3046615-murg	6	neos5	6	rococoB10-011000_dup1	6	timtab1	6
cod105_dup2	6	istanbul-no-cutoff	6	neos-3083819-nubu	6	neos859080	3	rococoC11-011100_dup1	6	toll-like	6
comp07-2idx	6	k1mushroom	6	neos-3216931-puriri	6	net12_dup1	6	roi2alpha3n4	6	tr12-30	6
comp21-2idx	6	lectsched-5-obj	6	neos-3381206-awhea	6	netdiversion	6	roi5alpha10n8	6	traininstance2	6
cost266-UUE	6	leo1	6	neos-3402294-bobin	4	ns1116954	1	roll3000	6	traininstance6	6
cryptanalysiskb128n5obj14	6	leo2	6	neos-3555904-turama	6	ns1208400	6	s100	6	trento1	6
cryptanalysiskb128n5obj16	6	lotsize	6	neos-3627168-kasai	4	ns1760995	6	s250r10	6	triptim1	3
csched007	6	mad	6	neos-3656078-kumeu	6	ns1830653	6	satellites3-25	6	uccase9	6
csched008	6	map10	6	neos-3754224-navua	6	ns1952667	6	satellites4-25	6	uct-subprob	6
cvs16r128-89	6	map16715-04	6	neos-3754480-nidda	6	nu25-pr12	6	savsched1	2	unitcal_7	6
dano3_3	6	markshare_4_0	6	neos-3988577-wolgan	6	nursesched-medium-hint03	6	sct2	6	var-smallemery-m6j6	6
dano3_5	6	markshare2	6	neos-4300652-rahue	6	nw04	1	seymour	6	wachplan	6
dws008-01	6	mas74	6	neos-4338804-snowy	6	opm2-z10-s4	6	seymour1	6		

Table A.1: The composition of the ground problem set of $\mathcal{D}$.

Name	Freq	Name	Freq	Name	Freq	Name	Freq	Name	Freq	Name	Freq
30n20b8	6	eil33-2	6	mik-250-20-75-4	6	neos-4738912-atrato	6	qap10	6	supportcase26	6
50v-10	6	eilA101-2	6	milo-v12-6-r2-40-1	6	neos-4763324-toguru	6	radiationm18-12-05	6	supportcase33	6
academictimetablesmall	6	enlight_hard	6	momentum1	6	neos-4954672-berkel	6	radiationm40-10-02	6	supportcase40	6
app1-2	6	fast0507	6	mushroom-best	6	neos-5052403-cygnet	5	rail01	2	supportcase6	6
assign1-5-8	6	fhnw-binpack4-4	6	n2seq36q	2	neos-5093327-huahum	3	rail02	6	supportcase7	6
atlanta-ip	6	fhnw-binpack4-48	6	n3div36	6	neos-5107597-kakapo	6	rail507	6	swath1	6
b1c1s1	6	gen-ip002	6	n5-3	6	neos-5188808-nattai	6	ran14x18-disj-8	6	swath3	6
bab2	6	gen-ip054	6	neos-1122047	6	neos-5195221-niemur	6	rd-rplusc-21	6	tbfp-network	6
bab6	6	germanrr	6	neos-1171737	6	neos-662469	6	reblock115	6	thor50dday	6
binkar10_1	6	glass-sc	6	neos-1354092	6	neos-873061	6	rmatr100-p10	6	timtab1	6
blp-ar98	6	glass4_dup2	6	neos-1445765	6	neos-911970	6	rmatr200-p5	6	tr12-30	6
blp-ic98	6	gmu-35-40	6	neos-1456979	6	neos-933966	6	rocI-4-11	6	traininstance2	6
bnatt400	6	gmu-35-50	6	neos-1582420	6	neos-957323	4	rocII-5-11	6	traininstance6	6
bnatt500	6	graph20-20-1rand	6	neos-2657525-crna	6	neos17	6	rococoB10-011000_dup1	6	trento1	6
bppc4-08	6	graphdraw-domain	6	neos-2746589-doon	6	neos5	6	rococoC11-011100_dup1	6	triptim1	3
brazil3	6	h80x6320d	6	neos-2978193-inde	6	neos859080	3	roi2alpha3n4	6	uccase9	6
buildingenergy	6	highschool1-aigio	6	neos-3004026-krka	5	net12_dup1	6	roi5alpha10n8	6	uct-subprob	6
chromaticindex1024-7	6	ic97_potential	6	neos-3024952-loue	6	netdiversion	6	roll3000	6	var-smallemery-m6j6	6
chromaticindex512-7	3	icir97_tension_dup1	6	neos-3046615-murg	6	ns1116954	1	s100	2	wachplan	6
cmflsp50-24-8-8	6	irish-electricity	6	neos-3083819-nubu	6	ns1208400	6	s250r10	6		
cms750_4	6	istanbul-no-cutoff	6	neos-3216931-puriri	6	ns1760995	6	savsched1	1		
co-100	6	k1mushroom	6	neos-3381206-awhea	5	ns1830653	6	seymour	6		
cod105_dup2	6	lectsched-5-obj	6	neos-3402294-bobin	4	ns1952667	6	sing326	6		
comp07-2idx	6	leo1	6	neos-3555904-turama	6	nu25-pr12	6	sing44	6		
comp21-2idx	6	leo2	6	neos-3627168-kasai	4	nursesched-medium-hint03	6	sorrell3	6		
cost266-UUE	6	lotsize	6	neos-3656078-kumeu	6	opm2-z10-s4	6	sp97ar_dup1	6		
cryptanalysiskb128n5obj14	6	mad	6	neos-3754480-nidda	6	peg-solitaire-a3	6	sp98ar_dup1	6		
cryptanalysiskb128n5obj16	6	map10	6	neos-3988577-wolgan	6	pg	6	square41	6		
csched007	6	map16715-04	6	neos-4300652-rahue	6	pg5_34	6	square47	6		
csched008	6	markshare_4_0	6	neos-4338804-snowy	6	physicianshed3-3	6	supportcase10	6		
cvs16r128-89	6	markshare2	6	neos-4387871-tavua	6	piperout-08	1	supportcase12	6		
dano3_3	6	mas74	6	neos-4532248-waihi	2	pk1	6	supportcase18	6		
dano3_5	6	mas76	6	neos-4647030-tutaki	6	proteindesign121hz512p9	6	supportcase19	1		
dws008-01	6	mcsched	6	neos-4722843-widden	6	proteindesign122trx11p8	6	supportcase22	5		

Table A.2: The composition of the ground problem set of both $\mathcal{D}^{8.9}$ and $\mathcal{D}^{8.11}$.

List of Algorithms

Bibliography

[AB08] Tobias Achterberg and Robert E. Bixby. Personal communication, 2008.

[AB09] Tobias Achterberg and Timo Berthold. Hybrid branching. In Willem-Jan van Hoeve and John N. Hooker, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 6th International Conference, CPAIOR 2009*, volume 5547 of *Lecture Notes in Computer Science*, pages 309–311. Springer Berlin Heidelberg, 2009. doi:10.1007/978-3-642-01929-6_23.

[ABCC95] David L. Applegate, Robert E. Bixby, Vašek Chvátal, and William J. Cook. Finding cuts in the TSP (A preliminary report). Technical Report 95–05, DIMACS, 1995.

[ACF07] Giuseppe Andreello, Alberto Caprara, and Matteo Fischetti. Embedding $\{0, 1/2\}$-cuts in a branch-and-cut framework: A computational study. *INFORMS Journal on Computing*, 19(2):229–238, 05 2007. doi:10.1287/ijoc.1050.0162.

[Ach07a] Tobias Achterberg. Conflict analysis in mixed integer programming. *Discrete Optimization*, 4(1):4–20, 2007. doi:10.1016/j.disopt.2006.10.006.

[Ach07b] Tobias Achterberg. *Constraint Integer Programming*. phdthesis, Technische Universität Berlin, 2007.

[AKM05] Tobias Achterberg, Thorsten Koch, and Alexander Martin. Branching rules revisited. *Operations Research Letters*, 33(1):42–54, 2005. doi:10.1016/j.orl.2004.04.002.

[AW13] Tobias Achterberg and Roland Wunderling. Mixed integer programming: Analyzing 12 years of progress. In Michael Jünger and Gerhard Reinelt, editors, *Facets of combinatorial optimization*, pages 449–481. Springer Berlin Heidelberg, 2013. doi:10.1007/978-3-642-38189-8_18.

[Bal71] Egon Balas. Intersection cuts—a new type of cutting planes for integer programming. *Operations Research*, 19(1):19–39, 1971. doi:10.1287/opre.19.1.19.

[BB12] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *J. Mach. Learn. Res.*, 13(10):281–305, 2012.

[BCC96] Egon Balas, Sebastián Ceria, and Gérard Cornuéjols. Mixed 0-1 programming by lift-and-project in a branch-and-cut framework. *Management Science*, 42(9):1229–1246, 1996. doi:10.1287/mnsc.42.9.1229.

[BCCN96] Egon Balas, Sebastián Ceria, Gérard Cornuéjols, and N. Natraj. Gomory cuts revisited. *Operations Research Letters*, 19(1):1–9, 1996. doi:10.1016/0167-6377(96)00007-7.

[BCMS90] Robert E. Bixby, Sebastián Ceria, Cassandra M. McZeal, and Martin W.P. Savelsbergh. An updated mixed integer programming library: Miplib 3.0., 1990. URL: https://hdl.handle.net/1911/101898.

[Bel61] Richard Bellman. *Adaptive Control Processes: A Guided Tour*. Princeton University Press, 1961.

[Ber06] Timo Berthold. Primal heuristics for mixed integer programs. Master's thesis, Technische Universität Berlin, 2006.

[Ber13] Timo Berthold. Measuring the impact of primal heuristics. *Operations Research Letters*, 41(6):611–614, 2013. doi:10.1016/j.orl.2013.08.007.

[BGG+71] M. Bénichou, J. M. Gauthier, P. Girodet, G. Hentges, G. Ribière, and O. Vincent. Experiments in mixed-integer programming. *Mathematical Programming*, 1:76–94, 1971. doi:10.1007/BF01584074.

[BGV92] Bernhard E. Boser, Isabelle M. Guyon, and Vladimir N. Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, COLT '92, page 144–152. Association for Computing Machinery, 1992. doi:10.1145/130385.130401.

[BH21] Timo Berthold and Gregor Hendel. Learning to scale mixed-integer programs. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(5):3661–3668, 2021.

[BHL+10] Timo Berthold, Stefan Heinz, Marco Lübbecke, Rolf H. Möhring, and Jens Schulz. A constraint integer programming approach for resource-constrained project scheduling. In Andrea Lodi, Michela Milano, and Paolo Toth, editors, *Proc. of CPAIOR 2010*, volume 6140 of *LNCS*, pages 313–317. Springer Berlin Heidelberg, 2010. doi:10.1007/978-3-642-13520-0_34.

[Bix12] Robert E. Bixby. A brief history of linear and mixed-integer programming computation. *Documenta Mathematica*, pages 107–121, 2012.

[BLBMT18] Radu Baltean-Lugojan, Pierre Bonami, Ruth Misener, and Andrea Tramontani. Selecting cutting planes for quadratic semidefinite outer-approximation via trained neural networks, 2018.

[BLP21] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: a methodological tour d'horizon. *Eur. J. Oper. Res.*, 290:405–421, 2021. doi:10.1016/j.ejor.2020.07.063.

[BLZ18] Pierre Bonami, Andrea Lodi, and Giulia Zarpellon. Learning a classification of mixed-integer quadratic programming problems. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, Lecture Notes in Computer Science, pages 595–604. Springer, 2018.

[BPL+17] Irwan Bello, Hieu Pham, Quoc V. Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning, 2017. arXiv:1611.09940.

[Bre96] Leo Breiman. Bagging predictors. *Machine Learning*, 24:123–140, 1996. doi:10.1007/BF00058655.

[Bre01] L. Breiman. Random forests. *Machine Learning*, 45:5–32, 2001. doi:10.1023/A:1010933404324.

[BS07] Ralf Borndörfer and Thomas Schlechte. Models for railway track allocation. In Christian Liebchen, Ravindra K. Ahuja, and Juan A. Mesa, editors, *7th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'07)*, volume 7 of *OpenAccess Series in Informatics (OASIcs)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2007. doi:10.4230/OASIcs.ATMOS.2007.1170.

[BS13] Timo Berthold and Domenico Salvagnin. Cloud branching. In Carla Gomes and Meinolf Sellmann, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 7874 of *Lecture Notes in Computer Science*, pages 28–43. Springer Berlin Heidelberg, 2013. doi:10.1007/978-3-642-38171-3_3.

[BZ78] Egon Balas and Eitan Zemel. Facets of the knapsack polytope from minimal covers. *SIAM Journal on Applied Mathematics*, 34(1):119–148, 1978. doi:10.1137/0134010.

[Cbc] Cbc. URL: https://www.coin-or.org/Cbc.

[CdAMJ17] Laura Calvet, Jésica de Armas, David Masip, and Angel A. Juan. Learnheuristics: hybridizing metaheuristics with machine learning for optimization with dynamic inputs. *Open Mathematics*, 15(1):261–280, 2017. doi:doi:10.1515/math-2017-0029.

[CF96] Alberto Caprara and Matteo Fischetti. {0, 1/2}-Chvátal-Gomory cuts. *Mathematical Programming*, 74(3):221–235, 1996. doi:10.1007/BF02592196.

[CMLW97] Cecile Cordier, Hugues Marchand, Richard Laundy, and Laurence A. Wolsey. bc-opt :A branch-and-cut code for mixed integer programs. Technical report, Université catholique de Louvain, Center for Operations Research and Econometrics (CORE), 1997.

[Com] Personal Communication. Dr. Timo Berthold, Sr Engineer, FICO Xpress Development Team, Fair Isaac Germany GmbH.

[Cpl] IBM CPLEX Optimizer. URL: https://www.ibm.com/analytics/cplex-optimizer.

[Dak65] Robert J. Dakin. A tree-search algorithm for mixed integer programming problems. *The Computer Journal*, 8(3):250–255, 1965. doi:10.1093/comjnl/8.3.250.

[Dat20] Leonid Datta. A survey on activation functions and their relation with xavier and he normal initialization, 2020. arXiv:2004.06632.

[Dem15] Mehmet Demirci. A survey of machine learning applications for energy-efficient resource management in cloud computing environments. In *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*, pages 1185–1190, 2015. doi:10.1109/ICMLA.2015.205.

[DHS11] John C. Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *J. Mach. Learn. Res.*, 12:2121–2159, 2011.

[EAB+20] Marc Etheve, Zacharie Alès, Côme Bissuel, Olivier Juan, and Safia Kedad-Sidhoum. Reinforcement learning for variable selection in a branch and bound algorithm. In Hebrard E. and Musliu N., editors, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research. CPAIOR 2020.*, volume 12296, pages 176–185. Springer, Cham., 2020. doi:10.1007/978-3-030-58942-4_12.

[ER18] Patrick Emami and Sanjay Ranka. Learning permutations with sinkhorn policy gradient, 2018. arXiv:1805.07010.

[FGW19] Stefan Fritsch, Frauke Guenther, and Marvin N. Wright. *neuralnet: Training of Neural Networks*, 2019. R package version 1.44.2. URL: https://CRAN.R-project.org/package=neuralnet.

[FJ18] Matteo Fischetti and Jason Jo. Deep neural networks and mixed integer linear optimization. *Constraints*, 23(3):296–309, 2018. doi:10.1007/s10601-018-9285-6.

[FM05] Armin Fügenschuh and Alexander Martin. Computational integer programming and cutting planes. In K. Aardal, G.L. Nemhauser, and R. Weismantel, editors, *Discrete Optimization*, volume 12 of *Handbooks in Operations Research and Management Science*, pages 69–121. Elsevier, 2005. doi:10.1016/S0927-0507(05)12002-7.

[Fou19] Robert Fourer. Software survey: Linear programming. *OR/MS Today*, 46(3), 2019. doi:10.1287/orms.2019.03.05.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[GDDM14] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pages 580–587, 2014. doi:10.1109/CVPR.2014.81.

[GGK+20] Prateek Gupta, Maxime Gasse, Elias B. Khalil, M. Pawan Kumar, Andrea Lodi, and Yoshua Bengio. Hybrid models for learning to branch, 2020. arXiv:2006.15212.

[GGNS21] Claudio Gambella, Bissan Ghaddar, and Joe Naoum-Sawaya. Optimization problems for machine learning: A survey. *European Journal of Operational Research*, 290(3):807–828, 2021. doi:10.1016/j.ejor.2020.08.045.

[GHG+21] Ambros Gleixner, Gregor Hendel, Gerald Gamrath, Tobias Achterberg, Michael Bastubbe, Timo Berthold, Philipp M. Christophel, Kati Jarck, Thorsten Koch, Jeff Linderoth, Marco Lübbecke, Hans D. Mittelmann, Derya Ozyurt, Ted K. Ralphs, Domenico Salvagnin, and Yuji Shinano. MIPLIB 2017: Data-Driven Compilation of the 6th Mixed-Integer Programming Library. *Mathematical Programming Computation*, 13:443–490, 2021. doi:10.1007/s12532-020-00194-3.

[GLP] GLPK. URL: https://www.gnu.org/software/glpk.

[GM21] Arunim Garg and Vijay Mago. Role of machine learning in medical research: A survey. *Computer Science Review*, 40:100370, 2021. doi:doi.org/10.1016/j.cosrev.2021.100370.

[Gom58] Ralph E. Gomory. Outline of an algorithm for integer solutions to linear programs. *Bulletin of the American Mathematical Society*, 64(5):275–278, 1958. doi:978-3-540-68279-0_4.

[Gom60] Ralph E. Gomory. An algorithm for the mixed integer problem. Technical report, RAND Corporation, 1960.

[Gur] Gurobi Optimization. URL: https://www.gurobi.com.

[HCP83] Ellis L. Johnson Harlan Crowder and Manfred Padberg. Solving large-scale zero-one linear programming problems. *Operations Research*, 31(5):803–834, 1983. doi:10.1287/opre.31.5.803.

[Hen18] Gregor Hendel. Adaptive large neighborhood search for mixed integer programming. Technical Report 18-60, ZIB, 2018.

[HJLT96] Thomas Hancock, Tao Jiang, Ming Li, and John Tromp. Lower bounds on learning decision lists and trees. *Information and Computation*, 126(2):114–122, 1996. doi:10.1006/inco.1996.0040.

[HMW19] Gregor Hendel, Matthias Miltenberger, and Jakob Witzig. Adaptive algorithmic behavior for solving mixed integer programs using bandit algorithms. In Fortz B. and Labbé M., editors, *Operations Research Proceedings 2018*, Operations Research Proceedings (GOR (Gesellschaft für Operations Research e.V.)), pages 513–519. Springer, 2019. doi:10.1007/978-3-030-18500-8_64.

[Ho95] Tin Kam Ho. Random decision forests. In *Proceedings of the Third International Conference on Document Analysis and Recognition*, volume 1 of *ICDAR '95*, pages 278–282. IEEE Computer Society, 1995.

[HR76] Laurent Hyafil and Ronald L. Rivest. Constructing optimal binary decision trees is NP-complete. *Information Processing Letters*, 5(1):15–17, 1976. doi:10.1016/0020-0190(76)90095-8.

[HWL+21] Zeren Huang, Kerong Wang, Furui Liu, Hui-ling Zhen, Weinan Zhang, Mingxuan Yuan, Jianye Hao, Yong Yu, and Jun Wang. Learning to select cuts for efficient mixed-integer programming, 2021. arXiv:2105.13645.

[JP82] Ellis L. Johnson and Manfred W. Padberg. Degree-two inequalities, clique facets, and biperfect graphs. In Achim Bachem, Martin Grötschel, and Bemhard Korte, editors, *Bonn Workshop on Combinatorial Optimization*, volume 66 of *North-Holland Mathematics Studies*, pages 169–187. North-Holland, 1982. doi:10.1016/S0304-0208(08)72450-2.

[JRT95] Michael Jünger, Gerhard Reinelt, and Stefan Thienel. Practical problem solving with cutting plane algorithms in combinatorial optimization. In William J. Cook, Laszlo Lovasz, and Paul Seymour, editors, *Combinatorial optimization*, volume 20 of *DIMACS Series in Discrete Mathematics and Computer Science, AMS*, pages 111 – 152, 1995.

[Kar72] Richard Karp. Reducibility among combinatorial problems. In Raymond Miller and James Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972. doi:10.1007/978-1-4684-2001-2_9.

[Kas80] G. V. Kass. An exploratory technique for investigating large quantities of categorical data. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 29(2):119–127, 1980. doi:10.2307/2986296.

[Kha79] Leonid G. Khachiyan. A polynomial algorithm in linear programming. *Doklady Akademii Nauk SSSR*, 244(5):1093–1096, 1979. doi:10.1016/0041-5553(80)90061-0.

[KLP17] Markus Kruber, Marco Lübbecke, and Axel Parmentier. Learning when to use a decomposition. In *Integration of AI and OR Techniques in Constraint Programming*, Lecture Notes in Computer Science, pages 202–210. Springer, 2017.

[KPP17] Daniel Karapetyan, Abraham Punnen, and Andrew Parkes. Markov chain methods for the bipartite boolean quadratic programming problem. *European Journal of Operational Research*, 260(2):494–506, 2017. doi:10.1016/j.ejor.2017.01.001.

[KRBA16] Jim Y. J. Kuo, David A. Romero, J. Christopher Beck, and Cristina H. Amon. Wind farm layout optimization on complex terrains–integrating a cfd wake model with mixed-integer programming. *Applied Energy*, 178:404–414, 2016. doi:10.1016/j.apenergy.2016.06.085.

[Kuh20] Max Kuhn. *caret: Classification and Regression Training*, 2020. R package version 6.0-86. URL: https://CRAN.R-project.org/package=caret.

[KvHW19] Wouter Kool, Herke van Hoof, and Max Welling. Attention, learn to solve routing problems!, 2019. arXiv:1803.08475.

[LBS84] Richard A. Olshen Leo Breiman, Jerome H. Friedman and Charles J. Stone. *Classification And Regression Trees*. Taylor & Francis Group, 1984. doi:10.1201/9781315139470.

[LD60] Ailsa H. Land and Alison G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520, 1960. doi:10.2307/1910129.

[Leg05] Adrien-Marie Legendre. *Nouvelles méthodes pour la détermination des orbites des comètes*. F. Didot, 1805.

[LLB+21] Eric Larsen, Sébastien Lachapelle, Yoshua Bengio, Emma Frejinger, Simon Lacoste-Julien, and Andrea Lodi. Predicting tactical solutions to operational planning problems under imperfect information, 2021. arXiv:1901.07935.

[LNL16] Minglei Li, Liangliang Nan, and Shaochuang Liu. Fitting boxes to manhattan scenes using linear integer programming. *International journal of digital earth*, 9(8):806–817, 2016. doi:10.1080/17538947.2016.1143982.

[Lod10] Andrea Lodi. Mixed integer programming computation. In Michael Jünger, Thomas M. Liebling, Denis Naddef, George L. Nemhauser, William R. Pulleyblank, Gerhard Reinelt, Giovanni Rinaldi, and Laurence A. Wolsey, editors, *50 Years of Integer Programming 1958-2008: From the Early Years to the State-of-the-Art*, pages 619–645, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. doi:10.1007/978-3-540-68279-0_16.

[Loh14] Wei-Yin Loh. Fifty years of classification and regression trees. *International Statistical Review*, 82(3):329–348, 2014. doi:10.1111/insr.12016.

[Lou15] Gilles Louppe. Understanding random forests: From theory to practice, 2015. arXiv:1407.7502.

[lps] lp_solve. URL: http://lpsolve.sourceforge.net.

[LT14] Andrea Lodi and Andrea Tramontani. Performance variability in mixed-integer programming. *TutORials in Operations Research*, pages 1–12, 2014. doi:10.1287/educ.2013.0112.

[LZ17] Andrea Lodi and Giulia Zarpellon. On learning and branching: a survey. *TOP*, 25(2):207–236, 2017. doi:10.1007/s11750-017-0451-6.

[LZMW09] Fuhai Li, Xiaobo Zhou, Jinwen Ma, and Stephen T. C. Wong. Multiple nuclei tracking using integer programming for quantitative cancer cell cycle analysis. *IEEE transactions on medical imaging*, 29(1):96–105, 2009. doi:10.1109/TMI.2009.2027813.

[Mar01] Alexander Martin. General mixed integer programming: Computational issues for branch-and-cut algorithms. *Lecture Notes in Computer Science*, 2241:1–25, 2001. doi:10.1007/3-540-45586-8_1.

[MBB+20] Seifeddine Messaoud, Abbas Bradai, Syed Hashim Raza Bukhari, Pham Tran Anh Quang, Olfa Ben Ahmed, and Mohamed Atri. A survey on machine learning in Internet of Things: Algorithms, strategies, and applications. *Internet of Things*, 12:100314, 2020. doi:10.1016/j.iot.2020.100314.

[M.D44] Joseph Berkson M.D. Applications of the logistic function to bioassay. *Journal of the American Statistical Association*, 39(227):357–365, 1944. doi:10.1080/01621459.1944.10500699.

[M.D51] Joseph Berkson M.D. Why I prefer logits to probits. *Biometrics*, 7(4):327–339, 1951. doi:10.2307/3001655.

[Mip] Miplib 2017 website. URL: https://miplib.zib.de/index.html.

[Mit02] John E. Mitchell. Branch-and-cut algorithms for combinatorial optimization problems. *Handbook of Applied Optimization*, pages 65–77, 2002.

[MK11] Kent McClymont and Edward C. Keedwell. Markov chain hyper-heuristic (mchh): An online selective hyper-heuristic for multi-objective continuous problems. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, GECCO '11, page 2003–2010. Association for Computing Machinery, 2011. doi:10.1145/2001576.2001845.

[MLIDLS14] Franco Mascia, Manuel López-Ibáñez, Jérémie Dubois-Lacoste, and Thomas Stützle. Grammar-based generation of stochastic local search heuristics through automatic algorithm configuration tools. *Computers & Operations Research*, 51:190–199, 2014. doi:10.1016/j.cor.2014.05.020.

[MM72] Robert Messenger and Lewis Mandell. A modal search technique for predictive nominal scale multivariate analysis. *Journal of the American Statistical Association*, 67(340):768–772, 1972. doi:10.1080/01621459.1972.10481290.

[MMWW02] Hugues Marchand, Alexander Martin, Robert Weismantel, and Laurence Wolsey. Cutting planes in integer and mixed integer programming. *Discrete Applied Mathematics*, 123(1):397 – 446, 2002. doi:https://doi.org/10.1016/S0166-218X(01)00348-1.

[Mos] Mosek. URL: https://www.mosek.com.

[MP43] W. S. McCulloch and W. Pitts. A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5(4):115–133, 1943.

[MS63] James N. Morgan and John A. Sonquist. Problems in the analysis of survey data, and a proposal. *Journal of the American Statistical Association*, 58(302):415–434, 1963. doi:10.1080/01621459.1963.10500855.

[NBG+20] Vinod Nair, Sergey Bartunov, Felix Gimeno, Ingrid von Glehn, Pawel Lichocki, Ivan Lobov, Brendan O'Donoghue, Nicolas Sonnerat, Christian Tjandraatmadja, Pengming Wang, Ravichandra Addanki, Tharindi Hapuarachchi, Thomas Keck, James Keeling, Pushmeet Kohli, Ira Ktena, Yujia Li, Oriol Vinyals, and Yori Zwols. Solving mixed integer programs using neural networks, 2020. arXiv:2012.13349.

[NVBB18] Alex Nowak, Soledad Villar, Afonso S. Bandeira, and Joan Bruna. Revised note on learning algorithms for quadratic assignment with graph neural networks, 2018. arXiv:1706.07450.

[Pad05] Manfred Padberg. Classical cuts for mixed-integer programming and branch-and-cut. *Annals of Operations Research*, 139(1):321 – 352, 2005. doi:https://doi.org/10.1007/s10479-005-3453-y.

[PAMAL15] Maria J. Pires, Pedro Amorim, Sara Martins, and Bernardo Almada-Lobo. Production planning of perishable food products by mixed-integer

programming. In João Paulo Almeida, José Fernando Oliveira, and Alberto Adrego Pinto, editors, *Operational Research*, volume 4 of *CIM Series in Mathematical Sciences*, pages 331–352. Springer, 2015. doi:10.1007/978-3-319-20328-7_19.

[PR91] Manfred Padberg and Giovanni Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM Review*, 33(1):60–100, 1991. doi:10.1137/1033004.

[Qui86] J. Ross Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986. doi:10.1023/A:1022643204877.

[Qui93] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., 1993.

[R C13] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, 2013. URL: http://www.R-project.org/.

[Ros58] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958. doi:10.1037/h0042519.

[Rud17] Sebastian Ruder. An overview of gradient descent optimization algorithms, 2017. arXiv:1609.04747.

[SCI] SCIP. URL: https://www.scipopt.org.

[SCZZ20] Shiliang Sun, Zehui Cao, Han Zhu, and Jing Zhao. A survey of optimization methods from a machine learning perspective. *IEEE Transactions on Cybernetics*, 50(8):3668–3681, 2020. doi:10.1109/TCYB.2019.2950779.

[SSBD14] Shai Shalsev-Shwartz and Shai Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, USA, 2014.

[SSR12] Ashish Sabharwal, Horst Samulowitz, and Chandra Reddy. Guiding combinatorial optimization with UCT. In Nicolas Beldiceanu, Narendra Jussien, and Eric Pinson, editors, *CPAIOR*, volume 7298 of *Lecture Notes in Computer Science*, pages 356–361. Springer, 2012.

[Sym] SYMPHONY. URL: https://projects.coin-or.org/SYMPHONY.

[TAF20a] Yunhao Tang, Shipra Agrawal, and Yuri Faenza.
learning to cut. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 9367–9376, 2020.

[TAF20b] Yunhao Tang, Shipra Agrawal, and Yuri Faenza. Reinforcement learning for integer programming: Learning to cut, 2020. arXiv:1906.04859.

[TGH11] Isaac Triguero, Salvador García, and Francisco Herrera. Differential evolution for optimizing the positioning of prototypes in nearest neighbor classification. *Pattern Recognition*, 44(4):901–916, 2011. doi:10.1016/j.patcog.2010.10.020.

[TMD+06] Sebastian Thrun, Michael Montemerlo, Hendrik Dahlkamp, David Stavens, Andrei Aron, James Diebel, Philip Fong, John Gale, Morgan Halpenny, Gabriel Hoffmann, Kenny Lau, Celia Oakley, Mark Palatucci, Vaughan Pratt, Pascal Stang, Sven Strohband, Cedric Dupont, Lars-Erik Jendrossek, Christian Koelen, and Pamela Mahoney. Stanley: The robot that won the darpa grand challenge. *J. Field Robotics*, 23:661–692, 2006.

[VC71] Vladimir N. Vapnik and Alexy Y. Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability & Its Applications*, 16(2):264–280, 1971. doi:10.1137/1116025.

[VFJ17] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks, 2017. arXiv:1506.03134.

[VRW87] Tony J. Van Roy and Laurence A. Wolsey. Solving mixed integer programming problems using automatic reformulation. *Operations Research*, 35(1):45–57, 1987.

[WB21] Jakob Witzig and Timo Berthold. Conflict Analysis for MINLP. *INFORMS Journal on Computing*, 33(2):421–435, 2021. doi:10.1287/ijoc.2020.1050.

[WBH21] Jakob Witzig, Timo Berthold, and Stefan Heinz. Computational aspects of infeasibility analysis in mixed integer programming. *Mathematical Programming Computation*, 2021. doi:10.1007/s12532-021-00202-0.

[WSC+16] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Łukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's neural machine translation system: Bridging the gap between human and machine translation, 2016. arXiv:1609.08144.

[Xpr] FICO Xpress Optimization. URL: https://www.fico.com/en/products/fico-xpress-optimization.

[ZJLB20] Giulia Zarpellon, Jason Jo, Andrea Lodi, and Yoshua Bengio. Parameterizing branch-and-bound search trees to learn branching policies, 2020. arXiv:arXiv:2002.05120.