Alma Mater Studiorum – Università di Bologna

DOTTORATO DI RICERCA IN

Computer Science and Engineering

Ciclo XXXIV

Settore Concorsuale: 09/H1 - SISTEMI DI ELABORAZIONE DELLE INFORMAZIONI

Settore Scientifico Disciplinare: ING-INF/05 - SISTEMI DI ELABORAZIONE DELLE INFORMAZIONI

METHODS FOR INTEGRATING MACHINE LEARNING AND CONSTRAINED OPTIMIZATION

Presentata da: Fabrizio Detassis

Coordinatore Dottorato
Prof. Davide Sangiorgi

Supervisore
Prof.ssa Michela Milano

Esame finale anno 2022

Abstract

This manuscript is the result of an industrial PhD done in collaboration with the University of Bologna and Optit srl, a company started as a spin-off of the Department of Industrial Engineering of the University of Bologna and active in the area of Operations Research (OR).
In the framework of industrial problems, the application of Constrained Optimization is known to have overall very good modeling capability and performance and stands as one of the most powerful, explored, and exploited tool to address prescriptive tasks. The number of applications is huge, ranging from logistics to transportation, packing, production, telecommunication, scheduling, and much more. The main reason behind this success is to be found in the remarkable effort put in the last decades by the OR community to develop realistic models and devise exact or approximate methods to solve the largest variety of constrained or combinatorial optimization problems, together with the spread of computational power and easily accessible OR software and resources.
On the other hand, the technological advancements lead to a data wealth never seen before and increasingly push towards methods able to extract useful knowledge from them; among the data-driven methods, Machine Learning techniques appear to be one of the most promising, thanks to its successes in domains like Image Recognition, Natural Language Processes and playing games, but also the amount of research involved.

The purpose of the present research is to study how Machine Learning and Constrained Optimization can be used together to achieve systems able to

leverage the strengths of both methods: on the one hand, this would open the way to exploiting decades of research on resolution techniques for COPs while, on the other hand, construct models able to adapt and learn from available data. In fact, the latter can result in a tremendous advantage, given that collecting data has now become a common, affordable practice in many industrial contexts.

In fact, the interplay between these two areas has drawn a lot of attention in recent years. In the first part of this work, we will survey the existing techniques and classify them according to the type, method, or scope of the integration; subsequently, we move to the main topic of the present research: integrating constraints in learning models. To this aim, we introduce a novel and general algorithm devised to inject knowledge into learning models by means of constraints, *Moving Target*. The method expands the existing techniques for constraint injection, with a relatively simple framework suitable to tackle non-differentiable and global constraints. In the last part of the thesis, two applications stemming from real-world projects and done in collaboration with Optit will be presented.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The field of Machine Learning has experienced an incredibly rapid growth in the last decades, thanks to unprecedented achievements in technologically important fields, such as Image Recognition, Natural Language Processing, Anomaly Detection and so on, as well as the combination of more efficient algorithms and widespread computational power.
These learning techniques were originally conceived to be purely data-driven, being able to abstract and build inner representations from mere data sets. Recently, more and more practitioners have started to investigate how to design more controlled learning models, able to combine the traditional knowledge extraction capability with (logic) rules and/or reasoning, allowing the user to inject domain knowledge into the modelling process as well as biasing the learning model towards a preferred behavior. Among the techniques that can be employed to this aim, many concepts are derived from constrained optimization, for instance regularization methods, Lagrangian relaxations, or data processing techniques where knowledge is represented in the form of constraints.

Another, complementary, line of work is that of exploiting the unstructured information extracted by such data-driven models within more complex systems. A recent trend is that of combining Machine Learning with Optimization processes, with the aim of devising systems able to build inner representation from external (and possibly mutable) data while preserving the possibility of a

precise mathematical formulation of the process at hand. In fact, data-driven model can play a major role in real-world applications of Operations Research models: for example, such a model can enter the modelling part of the process at hand, by replacing an analytical description with an inferred one. Or it can help the resolution process, replacing heuristic decisions with new ones, learned from historical data or by imitation of existing policies.

The two worlds of Machine Learning and Constrained Optimization, although conceived to tackle different tasks, i.e. predictive and prescriptive analytics, are more and more often used together. In this research work, we aim at understanding better the interplay, strengths and limitations of such integration. After surveying the more recent integration methods, we propose an algorithm to provide constraint support to learning models, by leveraging on the interaction between learning and constrained optimization. Finally, we present two practical examples that combine both predictive and prescriptive models to tackle real-world optimization problems.

1.1 Summary of Research Contributions

This thesis can be divided in three parts: in the first part, presented in Chapter 2, an overview of the existing techniques to integrate Machine Learning algorithms within optimization problems will be presented. We will divide the different methods according to the part of the optimization process in which they are involved.

- In Section 2.2.1 we review the methods aiding the formulation of the optimization problem.
- In Section 2.2.2 we analyze those methods exploiting a data-driven predictor to support the resolution process of an optimization problem.
- In Section 2.3 we list methods that belong to neither of the above, but rather use learning models to replace both the modelling and solving phase.

In Chapter 3 we present one of the major contribution of the present work, the Moving Targets algorithm. This latter expands the current set of methodologies to inject (hard) constraints in learning models, with a special focus on the type of constraints that are complex to tackle: 1) constraints without a

differentiable formulation and 2) global constraints, involving average values or, more generally, operators defined over large number of examples.

- In Section 3.3 the algorithm is presented.
- In Section 3.4 we provide a thorough comparison on existing methods, highlighting the major differences and range of applications.
- In Section 3.5 the algorithm is tested with a substantial empirical evaluation; benchmarks with existing methods will be presented.

In the last part, presented in Chapter 4, we move to the application of integration methodologies to a couple of real-world problems.

- In Section 4.1 a data-driven model will be used to estimate the failure's probability associated to components for predictive maintenance.
- In Section 4.2 we use a Machine Learning model to estimate the number of bins needed to pack a given set of items.

1.2 Preliminaries and Background

The content in this thesis builds upon several topics; in this Section we briefly review the theoretical background, although we will not give a detailed description but rather pointers to external resources. The reader should be acquainted with constraint and discrete optimization problems and their resolution algorithms. Furthermore, our contribution largely depends upon the theory of statistical machine learning, in particular the supervised learning setting. In Chapter 2, we will overview a large variety of methods involving, at different levels, optimization methods and machine learning algorithms. Although a deep understanding of the fields would be beneficial, a rough comprehension of the topics is sufficient to grasp the main ideas.

1.2.1 Optimization with Constraints

An optimization problem answers the question of which is the best possible solution among a set of candidates, given an objective function that measures the fitness of each candidate. Denoting by X the domain and assuming the fitness function is $f : X \to \mathbb{R}$, the problem is formulated as

$$x^* = \arg\min_{x \in X} f(x) \tag{1.1}$$

When the fitness function is convex over the domain X, the function is guaranteed to have only one minimum that is global; for this reason, it results easy to solve Equation (1.2), for instance by means of gradient-based techniques

$$x^{k+1} \leftarrow x^k - \alpha \nabla f|_{x_k} \quad , \quad \alpha > 0$$

Conversely, when the fitness function f is non-convex, the optimality guarantee does not hold and many local minima may exist, requiring a complete exploration of the domain region X. It is often the case that the process under examination can not be entirely expressed through Equation (1.2), because of additional requirements that the candidate solution must satisfy. These latter can be expressed as a set of constraints, i.e. relations between the variable involved; the subset of X in which all the constraints are satisfied is named feasible region and contains all the candidate solutions. Formally, a constrained optimization problem is defined as

$$x^* = \arg\min_{x \in X} \{f(x) | x \in C\} \tag{1.2}$$

Depending on the type of fitness function f, the constraints C and the variables involved, different paradigms emerge; we review the ones that will appear more frequently in this thesis.

Constraint Satisfaction Problem A Constraint Satisfaction Problem (CSP) is a triple $\langle X, D, C \rangle$ where X is the variable set, $X = \{x_1, \ldots, x_N\}$, $D = \{D_1, \ldots, D_N\}$ is the domain set, i.e. it specifies the possible values of each variable $x_i \in D_i$ and C is the constraint set. A constraint $c \in C$ is a pair (σ, ρ) where σ constitutes the list of variables involved and σ their relation; it is a subset of the Cartesian product of their domains. The constraint set C describes all the variable relations that must be satisfied by the solution to the problem. A variable assignment is a pair (x_i, a), which denotes that variable x_i has been assigned the value $a \in D_i$. Then, a solution of the CSP is a complete variable assignment $A = \{(x_1, a_1), \ldots, (x_N, a_N)\}$ that satisfies the constraints C. If the set of solutions is empty, then the CSP is unsatisfiable.

The CSP formulation is in fact powerful and can be used to tackle several problems in the fields of planning, scheduling, operations research, just to name a few. The resolution algorithms are usually based on the concepts of *inference* and *search*, rather than the brutal exploration of the domain region. Inference is related to *constraint progagation* and the idea is to eliminate portions of

the domain region that provably do not contain any solution. Depending on the arity of constraints, different propagation techniques may be used. Search, on the other hand, takes care of exploring the domain region, i.e. provides variable assignments, and combined with backtracking methods makes sure the exploration is complete. See [RVBW06] for a thorough overview.

Constrained Optimization Problem It is often the case that there exists a preference among the possible solutions of a CSP. If this is the case, then the most suitable language to describe our process is that of Operations Research (OR). This field is concerned with providing prescriptive tools, i.e. techniques to advise the best possible actions given a problem instance, usually expressed through a cost to be minimized and a set of rules to be satisfied. Although similar in scope to CSP, there exists a fundamental difference between Constraint Programming (CP) and OR: CP exploits constraints in a procedural way to act on the solution space, removing portion that provably do not contain solutions and thus iteratively constructing a solution. Conversely, in OR the feasible set is considered as a whole during the resolution process, with the resolution acting as a global search process. Formally, we define a Constrained Optimization Problem (COP) as

$$\begin{aligned} &\min_x f(x) \\ &\text{s.t. } g_k(x) \leq 0 \quad \forall k = 1, \dots C \\ &\qquad x_i \in D_i \quad \forall i = 1, \dots N \end{aligned} \tag{1.3}$$

where $x = (x_1, \dots x_N)$ represents the N decision variables, the objective function $f : X \to \mathbb{R}$ is real-valued, $g_i : x \to \mathbb{R}$ the C constraints and we specify the domains of the decision variable x. The combinatorial nature of the problem derives from the fact that the set of possible solutions is finite because (some of) the decision variables involved have integrality constraints. The objective function can be omitted if we are only interested in finding a feasible solution: in this case, we end up with a Constraint Satisfaction Problems (CSPs) but formulated in the OR language.

When the objective function f and the constraints g_k are linear function, and variables x_i are continuous, we obtain a *Linear Programming Problem* (LP). The problem has a finite set of solutions and is combinatorial, for the candidate solutions lie at the vertices of the polytope defined by the constraints. LP problems are rapidly solved with the Simplex Algorithm or variations of it. If

the objective function and the constraints are convex and the variables involved belong to continuous domains, Equation (1.3) becomes a *Convex Programming* Problem. Conditions sufficient to guarantee optimality are provided by the Karush-Kuhn-Tucker equations.
Until now we have considered only continuous decision variables; when we turn to integer decision variables, i.e. (some of) the domains D_i are discrete sets, new and more complicated classes of problems come into play. Linear programs with discrete domains sets are called *Integer Linear Programs* (ILP) and are NP-complete. When we have both integer and continuous variables, the associated problems are called *Mixed-Integer Linear Programs* (MILP) and similarly, when either the objective function or the constraints are non-linear we have *Mixed Integer Non-Linear Programs*. Good resources for Combinatorial and Convex Optimization are [PS98] and [BBV04], respectively.

Generally speaking, constrained optimization is founded on two milestones: *modelling* and *solving*. *Modelling* refers to the act of explicitly describing the problem, i.e. formulating the mathematical equations of each of its components as in Equation (1.3): this requires an expert to enumerate all the constraints that compose the model and (possibly) design a suitable objective function to properly rank each configuration. As it is usually the case, the modelling phase of a problem is in fact a reiterated process, where the components are refined until the problem's solution meets the expectation. This can result in a very time-consuming process, for instance because the system at hand is not simple to model or because of some unknown or hidden components.

On the other end, *solving* deals with the algorithmic resolution of the mathematical problem, and can range from exact methods to approximate ones, depending on the complexity of the problem. Intuitively, as the complexity of the model increases, one has to resort to approximate methods in order to tackle it efficiently, that is, within a reasonable amount of resources (being it time or computational power). In general, a fine-tuned balance between modelling and solving is required for the model to be effective and of practical use.

1.2.2 Machine Learning

Following [MRT18], Machine Learning (ML) can be intended as the set of computational methods that use experience to predict on unseen or unknown data.

The term experience may refer to multiple scenarios and is left undefined on purpose; in fact, ML techniques can employ different forms of information, for instance data sets, or active interactions with an environment. The grounding idea is to learn a statistical model from a data source and then use it to infer the most likely outcomes for new, unseen inputs.
The popularity of machine learning is mostly due to the success achieved by deep learning in the domains of image and speech recognition, natural language processing as well as playing games [Pou+18]. We will briefly introduce the most popular paradigms and refer to specific material for an in-depth review: for supervised and unsupervised learning, see [MRT18; GBC16; Bis07] and for Reinforcement Learning see [SB18].

Supervised Learning In the classical machine learning scheme we are provided with a data set $\mathcal{D}$ containing labelled input pairs $(x_i, y_i) \in \mathcal{D}$, and the task is to find the model that best describes the data, out of a family of available ones. The method lies in building a representation of the statistical distribution of the output variables in terms of the input ones, that can be later used for inferring values over the whole domain. This is an approximate representation, since the data set is just a sample of the true underlying distribution, $\mathcal{D} = (X, Y) \sim \mathcal{X} \times \mathcal{Y}$; in practical terms, a mathematical model is fitted to the set of data so as to minimize a predefined loss function, measuring the prediction error. This operation is termed *training*, or also *learning by example*, and it essentially consists of an optimization problem, usually solved via heuristic methods. Given the input and output data pairs, $x \in X$ and $y \in Y$, the learning problem can formally be stated as:

$$h^* = \arg\min_{h \in H} \{L(y^*, h(x))\} \tag{1.4}$$

where H is the set of available models, that is a function space of mappings $h : \mathcal{X} \to \mathcal{Y}$ and $L : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$ is a loss function. The loss function is usually a (semi) metric in the target space Y and defines how to measure the score of each model h. For example, a neural network model is represented as $h = h(\theta)$, with θ parametrizing the neuron connections, i.e. weights and biases. Depending on the target domain $\mathcal{Y}$ we distinguish between classification and regression tasks: in classification, the target domain is discrete and the purpose of the learning model is to assign the correct class to each given input. Conversely, in regression the target domain is continuous.

For a family of model parametrized by the vector θ and a training data set $\mathcal{D}$, Equation (1.5) becomes

$$h^* = \arg\min_{\theta} \frac{1}{|\mathcal{D}|} \sum_{(x_i,y_i)\in\mathcal{D}} L(y_i,\, h(x_i;\theta)) \tag{1.5}$$

which corresponds to minimizing the expected error over the input data set.

Semi-Supervised Learning In semi-supervised learning only a fraction of the available data is labelled. This is often the case when the labelling of input data is an expensive operation and data sets are not available: for example, in order to build a data set for image classification it is required to manually label each image, which can be very expensive when the data needed for training is in the order of (ten of) thousands examples. In semi-supervised learning the ML model receives both labelled and unlabelled data; the hope is that accessing the unlabelled data, the learner is able to reach performances superior than the ones obtained using solely labelled data.
As an extreme case, in unsupervised learning, the targets $y \in Y$ are not available and the learner is exclusively trained on the input. This kind of learning is usually aimed at finding underlying structures in the data; however, because of the absence of target labels, it is difficult to evaluate the performance of the learning model as well as to devise a suitable loss function for the task at hand. Examples of unsupervised learning problems are clustering problems, where a collection of data is grouped according to the similarity of their features, or methods involving component analysis, for instance Principal Component Analysis.

Reinforcement Learning Reinforcement learning is based on the idea of learning by interacting with an external environment. There are three key elements characterizing a RL problem: a learning agent connected with its surrounding environment, a possible set of actions with which it can interact and modify it and a goal to achieve. The learning paradigm differs from both supervised and unsupervised settings, for there is no labelled data but there is a quantitative reward to be maximized.

Reinforcement learning is usually framed as a Markov Decision Process, a structure that conveniently models sequential decision making processes. Broadly speaking, RL can be distinguished into two main schemes: *model-based* and

model-free learning. In the first scenario, we are given a restricted set of possible actions on the environment, while in the second case the model is completely free to devise how to interact with the environment. Moreover, *model-free* RL can be divided in diverse paradigms, depending on the object of the learning process. In *value-based* methods, the learner tries to estimate the expected reward associated to a given policy and action; the agent will then choose the action that maximizes the expected reward. Conversely, in *policy-based* methods, the agent's policy is parametrized and optimized by leveraging on the past experience, such that the final reward will be maximum.
The policy optimization problem can be formulated as

$$\pi^* = \arg\max_{\pi \in \Pi} \mathbb{E}\left[\sum_{t=0}^{T} \gamma^t \, R(s_t, a_t)\right] \tag{1.6}$$

where Π represents the space of policies, R is the reward obtained performing action a_t on state s_t and γ is a discount factor.
The field of Reinforcement Learning is vast and difficult to briefly summarize, therefore we refer to [SB18] for an in-depth overview.

1.3 Methodology

In the present thesis we overview many and diverse methods for the integration between ML models and Constraint Optimization Problems. Therefore, we introduce a methodological framework that will be used to support the description of such techniques; since the works that bring together the two fields can be very different, it can be difficult to abstract a general perspective on the integration. We find that a framework based on *transition systems* results very convenient, both to establish a high-level description of an optimization process and to effectively capture its components. Transition systems [BK08] are very popular in computer science since they establish a very powerful tool to describe dynamic systems; in particular, we adopt the notions of *states* and *transitions*. This notation, although usually employed in different contexts, allows us to highlight where and how learning models are effectively integrated in constrained optimization problems. For our purpose, that is surveying and classifying the recent trends in the field, we will not need an in-depth formalization; in fact, the methodology is loosely inspired by works such as [JM94], where transition systems are employed to abstract an operational semantics for Constraint Logic Programming, or [MR04], where authors devise a general

framework to describe metaheuristics under a multi-agent architecture, but departs from them for we do not aim at giving an exact representation of the process, but rather a qualitative description of it.

1.3.1 Optimization as a transition system

In the following, both the modelling and solving processes will be formulated by means of transition systems. While this choice is well-suited for the solving phase of a (combinatorial) optimization process, it results unusual for its modelling phase; however, subsequent examples on integration of learning components into the modelling of COPs will show the benefits of such formulation.
Transition systems are based on the concepts of *state* and *transition*: a *state* $\mathcal{S} = \langle A, B, C \rangle$ is a generic tuple of elements. A *transition* is a mapping between a state $\mathcal{S}$ and another state $\mathcal{S}'$ and will be denoted as $\rightarrow$, i.e. $\mathcal{S} \rightarrow \mathcal{S}'$. Transitions can be applied one or more times: when the number of a transition application is known to be n, it is indicated as $\rightarrow^n$, if it is not known, then it will be denoted with the symbol $\rightarrow^*$; if a transition has to be applied at least one time, it will be denoted as $\rightarrow^+$. A transition can also be performed as long as a stopping condition is not met: transitions of this kind are represented as $(\rightarrow)^c$, where c is the termination condition on the current state. Labels can be used to distinguish transitions performing different operations, for example $\underset{a}{\rightarrow}, \underset{b}{\rightarrow}, \ldots$. As for single transitions, sequences of labeled transitions can be applied several times as well, and this is denoted as before $(\underset{a}{\rightarrow}\underset{b}{\rightarrow})^n$.
This is loosely related to *Labelled Transition Systems* [BK08; VG01]: in LTS we have a pair $(\mathcal{P}, \rightarrow)$ with $\mathcal{P}$ a set of processes and $\rightarrow \subseteq \mathcal{P} \times \text{Act} \times \mathcal{P}$. Act is the set of possible actions that can be performed by the processes, for instance $p \xrightarrow{a} q$ means that by executing action a on process p, process q is reached.
In the following we will characterize the *states* and *transitions* that will be necessary to describe the modelling and solving processes of a COP.

Modelling process

A *model state* $\mathcal{M} = \langle X, f, g \rangle$ is a tuple of elements describing a problem instance (at a given time instant) through the following components: the set of decision variables $X = \{x_i : x_i \in D_i, i = 1, \ldots, M\}$ where D_i can have either infinite or finite cardinality, the objective function $f : \{X \times C_f \rightarrow \mathbb{R}\}$ and N constraints $g : \{X \times C_g \rightarrow \mathbb{R}\}$. Both the objective function and the constraints

are parameterized by the matrix $C = (C_f, C_g) \in \mathbb{R}^{(N+1)\times M}$ representing the coefficient of the decision variables X.
We can define a general, high-level, *refine* transition that operates on a model state $\underset{r}{\rightarrow}: \mathcal{M} \rightarrow \mathcal{M}'$ by modifying (some of) its components

$$\langle X, f, g\rangle \underset{r}{\rightarrow} \langle X', f', g'\rangle$$

The application of the refine transition can have multiple outcomes, that can be fully specified when it comes to each problem; examples of such operations are:

- adding a new constraint g_i
- adding a new decision variable x_i
- fixing the value of a decision variable $x_i = \bar{x}_i$
- modifying the loss function

Observe that this model state formulation allows to describe models with partial variable assignments, i.e. such that $X = \{X_i\}_{i\neq j} \cup \{\bar{x}_j\}$ where we denote by $\bar{x_j}$ a variable that has been assigned to a given value. Moreover, when all the variables have been assigned, i.e. $X = \bar{x}$ we are left with a complete assignment of the decision variables; the assignment $\bar{x}$ is a feasible solution of the model M iff $g(\bar{x}) \leq 0$. We represent a feasible solution as $S = \langle \bar{x}, f(\bar{x}), \mathit{true}\rangle$; in a minimization problem the solution is optimal iff $\forall S' \neq S,\ f(\bar{x}) \leq f(\bar{x}')$ (in a maximization problem we have instead $f(\bar{x}) \geq f(\bar{x}')$).
By denoting the optimal solution of a model M by S^*, we have

$$S^* = \left\langle \bar{x} = \arg\min_{x\in X} \{f(x) \mid g(x) \leq 0\}, f(\bar{x}), \mathit{true} \right\rangle$$

The act of modelling a constrained optimization problem is usually performed by an expert and can be seen as encapsulating the domain knowledge into a mathematically equivalent formulation. This process can be described within the transition system framework as a sequence of transitions that progressively refine the problem formulation. In particular, given the current model $\mathcal{M} = \langle X, f, g\rangle$, a target process specification $\mathcal{P}$ and the requirements it induces on the solution $\Omega(\mathcal{P})$, the modelling phase consists of a sequence of refine transition until $\mathit{sol}(\mathcal{M}) \neq \Omega(\mathcal{P})$.

We summarize the whole process of modelling as follows: first, an initial draft of the model is realized

$$\langle \emptyset, \emptyset, \emptyset \rangle \underset{r}{\rightarrow} \langle X, f, g \rangle$$

Then, it is iteratively refined until the solution meets all the process requirements, i.e.

$$\langle X, f, g \rangle \, (\underset{r}{\rightarrow})^{sol(\mathcal{M}) \neq \Omega(\mathcal{P})} \, \langle X', f', g' \rangle = \mathcal{M}$$

We introduce a new label for the transition $\underset{m}{\rightarrow} = (\underset{r}{\rightarrow})^{solution(\mathcal{M}) \neq \Omega(\mathcal{P})}$, which denotes the process of constructing an expert-designed model. This constitutes a special case of the refine transition and furnishes a compact way to represent the act of problem modelling

$$\langle \emptyset, \emptyset, \emptyset \rangle \underset{m}{\rightarrow} \langle X, f, g \rangle$$

Solving process

The scope of the resolution process of a COP is to explore the space of solutions searching for the one maximizing a *model scoring function* $\hat{z}$: this scoring function is identified with the (opposite of the) objective function of the optimization problem. The process can be formalized in terms of transition systems with the introduction of a resolution state $\mathcal{R} = \langle M, C, b \rangle$, consisting of the set of open model states to be processed $M = \{\mathcal{M}_i\} = \{\langle X, f, g \rangle_i\}$, the set of models under examination in the current resolution state C, with $C \cap M = \emptyset$ and the model state b with the best score $\hat{z}$ obtained so far.
The resolution process begins with a set of models M resulting from one or more modeling processes. For instance, a tree search-based resolution process would start from the singlet $M = \{\mathcal{M}\}$, whereas for a population-based resolution process $M = \{\mathcal{M}_1, \mathcal{M}_2, \ldots\}$. The initial state is then $\langle M, \emptyset, b = \arg\max_{\mathcal{M} \in M} \hat{z}(\mathcal{M}) \rangle$.

Once the resolution state is defined, we can introduce the following transitions:

- The *select* transition $\underset{s}{\rightarrow}: \mathcal{R} \rightarrow \mathcal{R}'$ chooses a set of models C to be processed according to a *select scoring function* $\hat{s}$ and removes it from the current pool M

  $$\langle M \supseteq C, \emptyset, b \rangle \underset{s}{\rightarrow} \langle M \setminus C, C, b \rangle$$

  where $C = \arg\max^{k}_{\mathcal{M} \in M} \hat{s}(\mathcal{M})$ is the set of the best k models according to $\hat{s}$.

- The *generate* transition $\underset{g}{\rightarrow}: \mathcal{R} \rightarrow \mathcal{R}'$ produces a number of model states from C by means of a *generating function* $\hat{g} : \{\langle X, f, g\rangle\} \rightarrow \{\langle X', f', g'\rangle\}$. Then, it adds them to the set of open model states M.

$$\langle M, C, b\rangle \underset{g}{\rightarrow} \langle M' = M \cup g(C), \emptyset, b\rangle$$

- The *evaluate* transition $\underset{z}{\rightarrow}: \mathcal{R} \rightarrow \mathcal{R}'$ evaluates the current model pool according to the model scoring function $\hat{z}$ and possibly updates the best model state b

$$\langle M, C, b\rangle \underset{z}{\rightarrow} \left\langle M, \emptyset, b' = \arg \max_{\mathcal{M} \in M \cup \{b\}} \hat{z}(\mathcal{M}) \right\rangle$$

The resolution process can be described with the formalism of transition systems as a sequence of *select*, *generate* and *evaluate* transitions. On a practical level, this means that the resolution process is completely defined by specifying the transition functions $\hat{s}$, $\hat{g}$ and $\hat{z}$ and can be described as $(\underset{s}{\rightarrow}\underset{g}{\rightarrow}\underset{z}{\rightarrow})^*$.

Complete exploration processes will end in a *resolution terminal state* $\langle \emptyset, \emptyset, b\rangle$ with b the best model state according to $\hat{z}$. The completeness of the resolution process can however be relaxed and for example replaced by a termination criteria based on a number N of iterations. In this scenario, a resolution terminal state will be the state $\langle M, C, b\rangle$ obtained after the sequence $(\underset{s}{\rightarrow}\underset{g}{\rightarrow}\underset{z}{\rightarrow})^N$.

Let us consider two practical examples: a tree search-based resolution algorithm and a population-based one. In a tree search-based resolution process, we start from a single model state and perform a binary branching operation on the current model to iteratively partition the solution space and eventually isolate the optimal solution. On the other hand, in a genetic algorithm we setup an heuristic exploration of the feasible region: starting from a pool of candidate solutions, we modify them at each iteration to guarantee a proper covering of the feasible solution space, while keeping track of the best solution found.

Tree Search

1. Initial resolution state starts from the model $\mathcal{M}$

$$\mathcal{R} = \langle M = \mathcal{M}, \emptyset, \emptyset \rangle$$

2. Select function $\hat{s}$ reflects the node exploration strategy

$$\hat{s}(\{\mathcal{M}\}) = \mathcal{M}$$

3. Generate function $\hat{g}$ is typically based on a binary branching strategy

$$\hat{g}(\{\mathcal{M}\}) = \{\mathcal{M}_1, \mathcal{M}_2\}$$

s.t. $int(\mathcal{M}) = int(\mathcal{M}_1) \cup int(\mathcal{M}_2)$ with $int(\cdot)$ denoting the sets of integer solutions obtainable from its argument

4. Evaluate function $\hat{z}$ corresponds to the model objective function.

Population-based heuristic

1. Initial resolution state starts from a solution pool $\{\mathcal{M}_i\}$

$$\mathcal{R} = \langle M = \{\mathcal{M}_1, \ldots, \mathcal{M}_N\}, \emptyset, \emptyset \rangle$$

2. Select function $\hat{s}$ isolates a subset of models that will be used to generate the new distribution

$$\hat{s}(\{\mathcal{M}_1, \ldots, \mathcal{M}_N\}) = M_s, |M_s| \leq N$$

3. Generate function $\hat{g}$ provides a new set of solutions, balancing between exploration and exploitation of the feasible region

$$\hat{g}(M_s) = \{\mathcal{M}'_1, \ldots, \mathcal{M}'_N\}$$

4. Evaluate function $\hat{z}$ corresponds to the model's objective function.

1.3.2 Supervised Learning as a transition system

As mentioned in Section 1.2.2, in Supervised Learning the act of training is usually performed via heuristic algorithms that iteratively refine the model's parameters until a stopping criterion is met. For example, Neural Networks employ gradient-based methods (SGD [Bot10], ADAM [KB14], ...), tree-based models construct partitions of the domain, SVM may involve either COPs algorithms as cutting planes [Joa06] or, more recently, gradient-based methods [SS+11].
The training process is based on a data, used to find the optimal configuration of the learner parameters, such that the chosen loss function is minimized (as

described in Section 1.2.2). Since the training data $\mathcal{D}$ is fixed, we can regard the process of learning as a pure optimization process, with the data information being contained in the loss function (that is precisely the objective function of the optimization problem).

At the beginning of the process, the machine learning model is specified: this is exactly a modelling process that results in an unconstrained optimization problem

$$\langle \emptyset, \emptyset, \emptyset \rangle \underset{m}{\rightarrow} \langle \theta, \ell, \emptyset \rangle$$

with

$$l_h(\theta) = \frac{1}{|\mathcal{D}|} \sum_{(x_i, y_i) \in \mathcal{D}} l\left(y_i, h(x_i; \theta)\right)$$

being l_h the learning model parameterized by θ. The resulting optimization process is described, as seen before, through a transition $(\underset{s}{\rightarrow}\underset{g}{\rightarrow}\underset{z}{\rightarrow})^c$ until a stopping condition c is satisfied (we usually set a limit on the iterations).

Since the learning process boils down to finding the optimal solution of the optimization problem described by θ and l, the resolution state representation is very simple, with the helper functions $\hat{z}$ and $\hat{s}$ being identities, while the generating function $\hat{g}$ acts on the model state $\mathcal{M}$ by refining the variable assignment:

- $g(\mathcal{M}) = \mathcal{M}'$ where $\mathcal{M}' = \langle \theta', \ell, \emptyset \rangle$ and θ' is updated via, for example, a gradient-based algorithm, $\theta' \leftarrow \theta - \alpha \nabla_\theta L$.

Chapter 2

Integration Approaches

This Chapter surveys a trend that has recently received a lot of attention: the integration of data-driven models in prescriptive problems. The reasons behind the ever-growing interest in the field can be reduced to two main factors: on one hand, the availability of data describing (part of) a process is an even more common factor, naturally leading to the employment of data-driven solutions. On the other hand, the recent successes of Machine Learning arose questions in the OR community on whether such models can be effectively employed to support existing algorithms and models.
In this Chapter we review the State Of The Art and classify the existing methods according to the type of integration and/or task of the Machine Learning model, by leveraging on the methodology described in Section 1.3. [1]

2.1 Motivation

When facing a decision involving multiple choices, it is very useful to carefully balance all the possible outcomes, taking into account the related costs and benefits and then choosing the best one. If such a problem can be mathematically formulated, by associating one (or more) variable to each possible outcome and a virtual "cost" to weight it, then an optimal solution can be

[1] The work presented in this Chapter has been done together with Luca Accorsi and is part of a manuscript still unpublished.

recovered, that is the mathematically most convenient one among all the possibilities. However, it is often the case that the formulation requires additional constraints, for the problem to resemble more realistically the original counterpart, or to prevent undesired outcomes. These constraints, upon proper translation into a mathematically consistent formulation, can be added to the mix and become part of the problem to solve. If the set of possible outcomes is finite, then we have a Combinatorial Optimization Problem.

In the framework of industrial problems, the application of constrained optimization is known to have overall very good performance and stands as one of the most powerful, explored, and exploited tool to address prescriptive tasks. The number of applications is huge, ranging from logistics to transportation, packing, production, telecommunication, scheduling, and so on. The main reason behind this success is to be found in the remarkable effort put in the last decades by the operations research community to develop realistic models and devise exact or approximated methods to solve the largest variety of constrained and/or combinatorial optimization problems, together with the spread of computational power and easily accessible OR software and resources.

2.1.1 Common Issues in Practice

The use of *off-the-shelf* combinatorial optimization models is often limited by the type of problem one encounters, especially when it comes to real-world scenarios.
There are many different reasons that can undermine the straightforward application of exact combinatorial models to the problem at hand, for instance:

(i) *Peculiarity*: the problem has system-specific parts that make the relative model not reducible to those present in the literature; an ad-hoc solution is then required. This is often the case in industrial problems, where specific behaviors and business-tailored solutions are required.

(ii) *Non-declarative structure*: some problem components of the problem do not have a declarative description, but need to be encoded in a mathematical model. For instance, a relation between variables is obtained via another model or a simulator.

(iii) *Complexity*: the model has a huge number of variables and/or constraints

and it requires a lot of computational time to be solved to optimality, thereby it has limited practical use.

In order to tackle these circumstances, the usual practice is to rely on heuristic algorithms. When the issue is the complexity or the peculiarity of the model, one can put aside *the* optimal solution and implement instead *greedy algorithms* to favor velocity over accuracy of the resulting solution or *meta-heuristics* for a better solution quality with an increase in computational resources; for instance, one can rely on local search approaches, or backtracking-based methods until a good solution is found. A slightly different method is to introduce heuristic elements in the model itself, based on how the modeler expects the solution to be. This can be done by relaxing a few constraints of the exact model, or by simplifying the objective function. The method also applies to problems lacking a declarative description, through the introduction of approximated heuristic elements. All these methods have the advantage that, despite the solution being sub-optimal, the model becomes of real use and has usually good computational scalability. Unfortunately, there are some major drawbacks, as well: apart from being approximated models, heuristic approaches result very problem-specific, lacking any sort of generality, and they may require a certain expertise to be efficient and provide satisfactory results.

2.1.2 A recent trend

In recent years, a growing number of works have faced the possibility of employing methods coming from the automatic learning field in combinatorial optimization. Indeed, it is nowadays established the role of machine learning as a major tool to address data-driven problems, with the technology providing a huge amount of data and learning models the tools for extracting useful knowledge from them. The spread and availability of sensor technology, combined with a pervasive IT infrastructure, allow private companies to gather quantitative information describing the functioning of their processes, which can be turned into valuable knowledge via the proper learning tools. At the same time, the recent advancements and unprecedented achievements of machine learning, especially when it comes to neural networks, lead to thinking about potential applications even to complex problems, as is the case of most combinatorial optimization models.
The two worlds of machine learning and combinatorial optimization, which we may call *predictive* and *prescriptive analytics*, are always more frequently used in similar - if not identical - contexts, although for different tasks. On the

one hand, machine learning algorithms are well-suited to construct statistical representation from examples, fostering the analysis of large amounts of data while giving up on an exact control of the outcome; on the other hand, optimization problems are based on strict mathematical modeling of the given process, that answer to specific requirements and can (ideally) be solved exactly.

In this Chapter we aim at surveying the existing research directions and organizing the established methods in a schematic manner, to better express the interaction between the two fields.

2.2 Integration Schemes

In Figure 2.1 we identify the main techniques of how combinatorial problems can leverage learning algorithms. Generally speaking, learning algorithms can

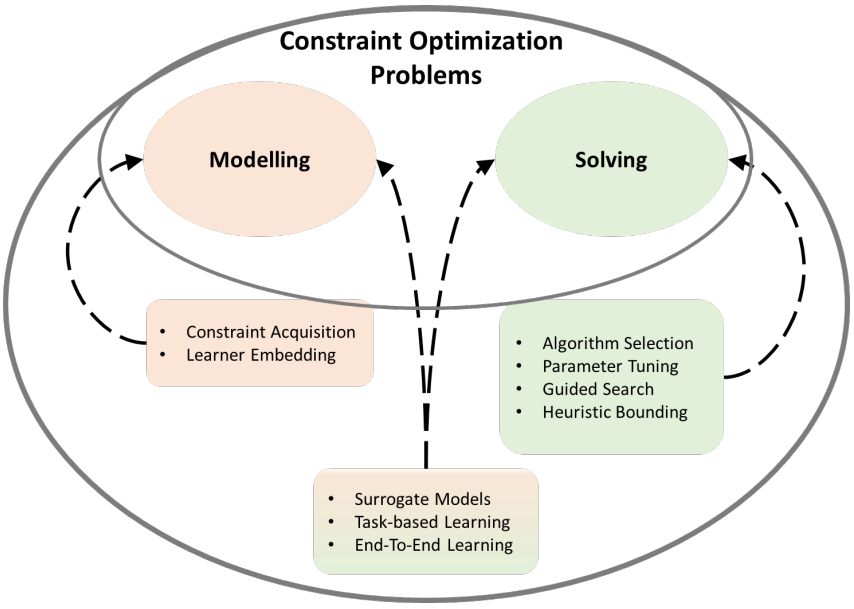

Figure 2.1: Integration schemes to boost combinatorial problems with learning.

be employed both in the modeling and solving phases of a COPs. In the first case, learning can be used to model (part of) the constraints [BS12; DRPT18; Lal+10; PK17], the objective function, or both[Kum+21]; for instance by making use of historical data, examples, or sources of external knowledge. It can

also be used to learn a feasible mathematical formulation of the problem: this is achieved with *constraint acquisition* algorithms [Bes+17], where a model is built from scratch by repeatedly stacking constraints consistent with a set of solutions.
As for the solving phase, the central question is whether it is possible to exploit a learner for improving the resolution of a given problem. We can classify the methods on the basis of the task the learner is required to do:

- In *Algorithm Selection*, a learner is used to select from a pool of algorithms the best performing one.
- A similar setting is found in *Parameters Tuning*, where learning is used to choose the most promising configuration of an algorithm, once this latter has been selected.
- *Guided Search*, where a learner is used to guide the exploration of the variable configuration space, indicating the most promising direction of search.
- *Bounding*, where a learner estimates the cost associated with a partial variable configuration in order to speed up the exploration of the feasible solution space.

There exist other methods that can be identified neither with modeling nor with solving, because they combine both categories. For example, *surrogate models* are used in *black-box optimization* to provide a mathematical approximation of the true underlying problem structure, when this latter does not have a clear algebraic formulation, or we do not know it, but an oracle is accessible and can be queried (for instance a simulator). In this scenario, we could be able to evaluate the objective function by interacting with the simulator and hence construct a surrogate function based on such outcomes. The surrogate mimics the original system and makes it possible to devise an approximate model for the process at hand. Machine learning models are in fact an example of surrogate models, although the latter usually come with a more suitable algebraic form.
Task-based learning is similar to what is done when modeling part of the COP with machine learning, with the fundamental distinction that here the learner is not trained beforehand, but instead, it is updated while solving the optimization problem. In this sense, such methods lie in between modeling and solving.

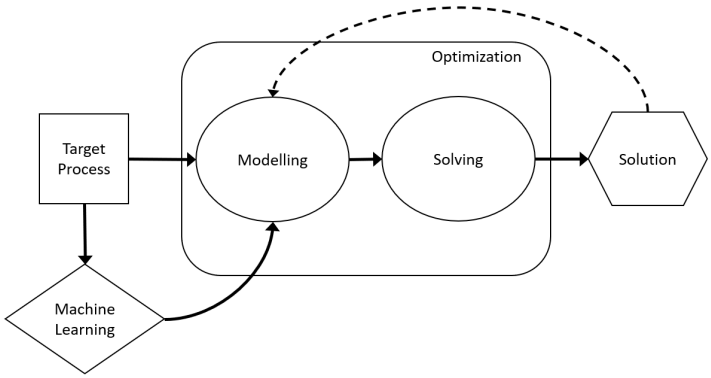

Figure 2.2: Machine learning is used to boost the modeling phase of an optimization problem.

Another way to exploit a learner to solve a COP and bypass entirely its explicit formulation is to use *reinforcement learning*. A learner can be used to solve graph-based combinatorial problems (for instance Travelling Salesman Problem or Vertex Coloring) by embedding the problem's structure within a neural architecture. In general, this requires thorough crafting but has the potential of scaling to different instances and, even more interesting, to multiple instance sizes. The key idea is to inject the learner with external knowledge of the combinatorial structure of the problem, making it implicitly aware of the existing constraints: the combinatorial structure of the problem is introduced in the learner by piloting its loss function.

In the next Sections we overview each integration scheme and summarize the main works pertaining to it.

2.2.1 Modeling

As mentioned before, the modeling phase of a COP requires a thorough knowledge of the domain in order to formulate a model that is representative of the problem at hand. To aid the modeling task, part of the declarative process can be demanded to learning models, as depicted in the integration scheme in Figure 2.2. The expert might in fact be replaced by an algorithm able to learn from data: this comes with a double advantage. On one hand, it speeds up the modeling phase, provided a good data source is available; on the other hand, it

allows to measure the descriptive performance of a model, that is how good the model represents the specific process. In the case of a machine learning model, we may measure its performance by simply looking at the loss function, or estimating the prediction error. On the downside, such a parametric description of (part of) the model can lead to difficulties: first of all, the learner may come with complex mathematical functions, making the optimization problem hard to solve. Furthermore, the learned relations heavily depend on the data used in the training phase; such data has to be representative of the process at hand, for the COP to be effective and generalize well on unseen scenarios.

Constraint Learning

Several works focus on the problem of learning constraints from data in an inductive learning framework, that is construct a constraint theory (or constraint network) from examples, see [DRPT18] for an overview. This learning paradigm can be used not only for predicting the satisfiability of unknown instances, but also for completing partial instances, or optimizing the completion when a loss function is given. Constraint learning can either address hard satisfaction problems, e.g. SAT or CSP problems, or soft constraints, for instance, MAXSAT or problems with preferences over constraints.

The representation of constraints can be Boolean or First Order Logic, and the method either leverages active or passive learning. In passive learning, the algorithm is provided with already collected examples, as is the case of historical data, which imposes an upper bound to the available knowledge. On the other hand, in active learning it can interact with an external oracle, either human or simulated, and query it with a specific configuration; this enables the algorithm to actively guide the exploration towards the domain regions that appear to be most promising, e.g. verifying the constraints that result more convenient for constructing the constraint theory. In this case, the problem of acquiring constraints can be regarded as an interactive process, with the learner trying to formulate the best representation of the problem through the oracle.

In [BS12] the authors propose Model Seeker, an approach to extract global constraints from positive examples of structured problems. They rely on Constraint Seeker, a tool for ranking the best matching constraints, given an input sample, and an existing library for global constraints as a learning bias, i.e. the global constraint catalog. The approach is well suited for problems with an underlying structure, as it involves different transformations of the problem

data to identify recurrent patterns; the use of a fixed bias gives an advantage over methods that construct it from scratch, although global constraints can be expensive to be verified.

In [Lal+10] a constraint acquisition approach based on inductive logic programming (ILP) is presented. The goal is to acquire an abstract description of the model, resulting better than other constraint acquisition methods, for example, CONACQ, in that it can learn from mere examples rather than solutions to the original problem. In the learning phase, the algorithm needs essentially three things: positive examples, negative examples, and background information. Starting from this knowledge and by means of a rule-based language, it learns the CSP definition that best discriminates between solutions (positive examples) and non-solutions (negative examples).

In the field of active learning, we find QUACQ [Bes+13], an algorithm able to ask the master partial queries and use these to construct a constraint network. A partial query is an assignment to only a subset of the problem variables. A similar setting is presented in CONACQ [Bes+17]: the basic idea is to build a bias, which comprises all the possible constraints that can be built out of the problem's variables and the admitted constraint relations. The algorithm then proceeds with the validation of constraints by interacting with the user, thus building the problem's constraint network.

A different line of work is to use an exact formulation to learn constraints all at once, by solving a proper optimization problem. In [PK17] a MILP formulation is used to generate constraints with a definite algebraic form from constraint synthesis problems, that is, a list of potential variable assignments and the corresponding feasibility. The usage of an optimization problem allows total control over the algebraic form of the constraints to generate and guarantees their satisfaction while giving free choice on the objective function to minimize: by choosing to minimize the number of terms used in the constraints, we can produce more compact and human-readable constraint sets. As a major drawback, the scaling rapidly becomes an issue as the number of variables grows.

In [Kum+20] the authors present a technique to learn (part of) a MAX-SAT Optimization Problem from data, but explicitly take into account the context of each example. A context can be intended both as a partial variable assignment or as a set of constraints that influence the outcome of an optimization problem. This feature is often underestimated by learning methods but can lead to substantial sub-optimalities when the context changes at inference time, or in scenarios where a given assignment is discarded either for being

infeasible or sub-optimal. After showing that Empirical Risk Minimization can be used to learn low-risk MAX-SAT problems for any context, the authors introduce HASSLE: the method leverages a MILP formulation to learn the parameters of soft and hard constraints of the MAX-SAT, by providing a tight approximation to the original learning problem. Besides, by parameterizing both hard constraints and the objective function with soft constraints, the method is capable of learning the two of them at training time.

Generally speaking, the learning of constraints from data is firmly grounded on symbolic representations of the rules, whose parameterization is learned from data. A sub-symbolic alternative is presented in [Cir+20], where the set of learnable constraints is modeled by means of neural networks. A set of task functions is learned from data, with the additional complexity of a constraint regularization. Then, the acquisition of new constraints boils down to maximizing the transfer of information between them and the task functions: the transfer is modeled via the Mutual Information method. A general framework is considered: given a classification problem together with a set of pre-defined constraints, a set of task functions (i.e. one for each possible outcome class), and a set of learnable constraints, the problem is split into three different levels: input data constitutes the input space, task functions define the concept space (containing high-level functions) whereas the constraints belong to the rule space, that embeds both historical and new knowledge. In order to interpret the sub-symbolic rules learned by the neural network, the resulting network is interpreted as a logic network, reducing its predictions to First-Order Logic formulas. This constitutes the bridge between the sub-symbolic learning technique and standard constraint theories, formulated through First Order Logic. It is interesting to notice that, in addition to extracting useful knowledge from the tasks, the new constraints act as a regularization of the former, impacting their generalization skills. The method beats the baseline (uninformed) learner, especially when the number of samples is low, although the scaling to a large number of constraints results computationally very expensive.

Constraint acquisition processes as transition systems

We can describe the process of constraint acquisition within the transition systems framework as follows:

1. Start from the empty model state

$$\mathcal{M} = \langle \emptyset, \emptyset, \emptyset \rangle$$

2. Build the bias set $\mathcal{B}$ of all the possible constraints, either from the training data set $\mathcal{D} = \{x_i\}$, $i = 1, \ldots, N$, or by exploiting an existing constraint dictionary

3. Build the constraint set, i.e. $\forall x_i \in \mathcal{D}$

   (a) check the set of constraints $\{g_j\}_i \subseteq \mathcal{B}$ compatible with x_i (either via a generating procedure or extracting them from a bias set) and involving the decision variables z_i

   (b) Add the newly generated constraints and variables

$$\langle X, 0, g \rangle \underset{r}{\rightarrow} \langle X \cup \{z_i\}, 0, g \cup \{g_i\} \rangle$$

   (c) Prune constraints $\{\hat{g}_i\}$ that are no longer compatible with the training examples seen so far

$$\langle X, 0, g \rangle \underset{r'}{\rightarrow} \langle X, 0, g \setminus \{\hat{g}_i\} \rangle$$

We can summarize the process with a single transition

$$\langle X, 0, \emptyset \rangle \, (\underset{r}{\rightarrow}\underset{r'}{\rightarrow})^N \, \langle X, 0, g \rangle$$

that takes the place of the modeling transition $\underset{m}{\rightarrow}$; note that we have as many atomic transitions as the number of training examples.

Learner embedding

The modeling phase of a combinatorial problem can be boosted by embedding a learner directly in the definition of the problem. The learner can be employed to model relations that are unknown or better modeled from data, for instance in the case of functions that evolve over time. Or else, the relations could be better handled by learning models, because of their complexity - think for example to hydrodynamic systems, where a physical description exists but is computationally intractable. In all the cases where there is available data describing part of the problem and a machine learning model that fits well the

data, we might be interested in replacing the formal definition of the relations with the model: the embedding can be done either in the form of a constraint or within the objective function.
This is what is done in [BLM15; LG16; LMB17], where the authors provide examples of how to directly insert pre-trained learning models (decision trees and neural networks) in the form of constraints within CP and MILP, preserving the original language of the optimization problem. For instance, in [BLM15] a decision tree is used to approximate a complex cost function and embedded in a CP model. The training set is composed of a set of attributes representing the dependence between decision variables and cost function. The trained decision tree h^* can be embedded into a CP by encoding each path root to leaf as a set of constraints $\Xi(h^*) = g'$ containing a conjunction of disjunctions associated with the attributes and values of the decision tree nodes.
Although with a different aim, this is similar to what is done in [AAV19], where the authors devise a fair classifier by encoding a decision tree within a MILP and adding an unfairness penalty to the objective function of the problem. In [FJ18] a feed-forward artificial neural network with ReLU activation functions is inserted in a MILP, with the goal of probing the robustness of the network at image classification.
These approaches do not lack drawbacks: only a fairly simple model (e.g linear or piece-wise linear) can be inserted in an optimization problem without making it nearly impossible to solve. Moreover, embedding a fully-fledged model usually requires the introduction of many variables and (integrality) constraints, which can again result in an overkill for the solver.
Another recurrent problem, especially when combining learners with the objective function, lies in the fact that the learner's uncertainty is not taken into account by the optimizer. This might result in the optimizer ending in regions of the configuration space for which the learner hasn't been trained properly, with very poor outcomes. In the upcoming subsections, we will overview methods that try to overcome these limitations.

Embedding of a learner as transition systems

Suppose we are given a trained machine learning model h^*, that we can represent as a mapping on the feature space, $h^* : \mathcal{X} \to \mathbb{R}$. The learning model can be embedded (or encoded [a]) into a combinatorial problem by introducing a set of constraints $\hat{g}$ generated by its embedding (or encoding) $\Xi(h^*)$. This can be defined in our transition system as follows.

1. First, the model is designed by an expert

$$\langle \emptyset, \emptyset, \emptyset \rangle \underset{m}{\rightarrow} \langle X, f, g \rangle$$

2. Then, the trained machine learning model h^* is inserted into the model

$$\langle X, f, g \rangle \underset{r}{\rightarrow} \langle X, f, g \cup \hat{g} \rangle$$

by means of the embedding in terms of host language elements

$$\hat{g} = \Xi(h^*)$$

3. Finally, the resolution process remains unaltered.

[a] We refer to encoding when the machine learning model can be directly formulated in the host language of the problem, whereas we use the term embedding when it is required to define an operational semantics along with the encoding

2.2.2 Solving

The resolution phase of an optimization problem is the core part of any optimization process and its study still draws a lot of attention. The first algorithms trace back to the '40s, when several independent statements of the LP problem, as well as resolution approaches, appeared: Dantzig's famous Simplex Algorithms was invented in 1947, and published in 1951 [Dan51; Dan90] and shortly after, in 1960, the branch-and-bound algorithm was introduced [LD60]. This latter opened the way for Mixed-Integer Linear Programming (MILP) problems. The work faced the problem of dealing with discrete variables in LP problems, by means of a divide-and-conquer strategy: the search in the feasible space is boosted by repeatedly partitioning the original space into smaller and smaller subsets, by means of bounding and pruning, which makes the problem easier to tackle.
The novel algorithms provided a fast way to solve problems that would otherwise be intractable while retaining a good level of modeling, which lead to a rapid expansion of the field both in academic and industrial environments. In the subsequent years, new solving algorithms were introduced, extending the range of applicability to Quadratic Programming (QP) [NW06], Convex Programming (CP), [BBV04] and also Mixed-Integer non-Linear Programming [PS98].

However, except for the simple case of LP problems, the research on resolution algorithms is still ongoing: in fact, even well-established algorithms such as Branch-And-Bound are partially guided by heuristic choices, usually backed by empirical evidence. For example, in B&B the decision on the variable to branch on, or the policy to adopt for selecting the node to explore are usually based on experience. Notice that, although these choices will affect the performance of the resolution process, they don't undermine the optimality of the solution. In fact, the incorporation of heuristic arguments can speed up the exploration of the feasible space while preserving its completeness.

In the last decade, thanks to the popularity gained by ML, there have been many attempts at exploiting learning methods to boost resolution algorithms, as it is summarized in Figure 2.1. A general representation of such integration is given in Figure 2.3: the learning algorithm can be fed with information about the target process and features from the resolution algorithm. At inference time, the trained algorithms will be employed by the resolution process. In this section, we will focus on such works.

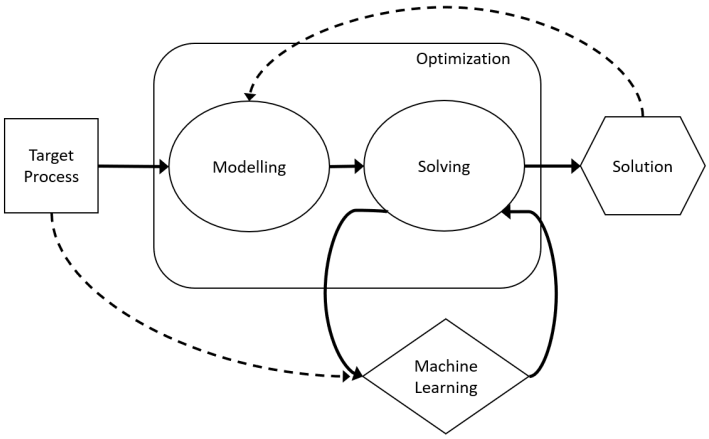

Figure 2.3: Machine learning is used to boost the solving phase of an optimization problem.

Search guidance

Guiding an exact or heuristic resolution process by means of offline or online generated knowledge consists of a natural way to integrate machine learning approaches within the more traditional COP resolution process. However, to be effective and of practical use, the two approaches must be carefully balanced. Offline learning can be used to extract rules that are then hard-encoded into the search process. As an example, consider [AS19], in which a graph-based COP is solved by using a meta-heuristic approach: a few simple rules, derived from a preliminary analysis with a decision tree classifier, are used to change the value of some arcs to improve the process of escaping from local optima. This is the simplest possible setting in which machine learning supports a resolution process without any overhead at runtime. Moreover, the preliminary analysis cost is amortized over all the future resolutions, when the predictor provides useful knowledge. However, for the derived rules to be meaningful, the training data must be representative of all possible instances, i.e. instances must be related to the training distribution. Unfortunately, this is very unlikely in COPs since even ad-hoc algorithms tend to behave very differently on instances with a different structure or scale.
Learning, either online or offline, can be combined with a runtime model inference. A seminal work not based on ML algorithms is presented in [KNS09]: the authors resort to a restarting policy to exploit incomplete explorations of the search tree of a binary MILP and guide the successive branching decisions. At each fathomed node, the branching decisions that lead to the fathoming are identified by generating a corresponding minimum cardinality clause.

Many researchers focused on the problem of branching [ALW17; Kha+16; HDIE14; LZ17; MAWL16; Gas+19]; we summarize the main elements of these works in Table 2.1. The framework is usually that of supervised learning, where a model is trained to replace the desired branching policy. In [ALW17] a B&B algorithm is placed side by side with a learning component imitating a Strong Branching policy: during the branching process, the learner identifies the variable to branch on by assigning a *strong branching score* to each fractional variable and picking the most promising one. The data set on which the learner is trained is built by optimizing a number of training problems with a B&B and strong branching. At each B&B node, a fixed number of features $\Phi(x)$ are used to characterize fractional variables x and a strong branching score s_x is computed by performing strong branching on them. The data set thus consists of pairs $(\Phi(x), s_x)$ for each fractional variable x found during the B&B execution

on the training problems. A similar setting is devised in [Kha+16], but ML is employed in a *learning-to-rank* fashion, with a SVM ordering the branching variables according to their expected outcome. Moreover, the learner is used in an online setting: at the beginning of the resolution process data is collected and used to train the learner; then, the algorithm switches to inference mode replacing the original Strong Branching policy with the learned one. In this way, the algorithm is also able to adapt to different instances *on the fly*. The problem of *node selection* in B&B is tackled in [HDIE14]: here ML is used to select both the most promising node to be explored and to decide whether to prune it or not, thus effectively guiding the search process. The training data is constructed by taking as a label the node containing the optimal solution, and the goal of the learner is to predict the best policy (i.e. choose the best node to expand), given a set of candidates. Since the policy corresponds to a sequence of explored nodes and thus the error propagates through the learner's decisions, errors close to the root node are weighted more. The idea of learning a policy to guide the exploration is further investigated in [Gas+19]: the authors parametrize a variable selection policy with a Convolutional Neural Network that acts on an encoding of the current state of the B&B process. Such encoding is given by a graph that represents the relational dependence between the variables and constraints of the problem, enriched with additional features.

	Learner	Task	Strategy
[ALW17]	XRT [2]	Regression	Variable Selection
[Kha+16]	SVM [3]	Learning to rank	Variable Selection
[HDIE14]	SVM [3]	Policy Learning	Node Selection / Pruning
[Gas+19]	GCNN	Policy Learning	Variable Selection

Table 2.1: Examples of integration of ML models within a Branch & Bound algorithm

Learning can also be used to decide when to perform expensive operations, for example as done in [KLP17]: here a predictor is used to decide whether to perform a Dantzig-Wolfe decomposition. In [Kha+17] authors employ learning to decide when and where to run primal heuristics in MIP problems, given the trade-off between advantage and complexity. However, online learning may

[2]Extremely Randomized Tree [GEW06]
[3]Support Vector Machines [CV95]

not be suitable for contexts in which the resolution process is too short or has time constraints, as it can introduce a computational overhead that significantly impacts the process. Moreover, one has to keep in mind that the generated knowledge might produce decisions strongly biased by the history of search that lies in the training data. This can have a dramatic effect when switching from different class of MILPs: an attempt to devise a method capable of generalizing the problem of branching variable selection across multiple problems is proposed in [Zar+20]. A DNN is used, together with a set of general features, to learn a branching policy that also considers the actual state of the branching tree.

Querying the model during the search process requires the careful balancing of state accuracy and prediction meaningfulness with resolution speed. On the one hand, the representation associated with the current exploration state must be expressive enough to allow for meaningful decisions. However, this contrasts with the speed requirements needed for being competitive with traditional resolution algorithms. In fact, the computational effort for computing such representations (and for keeping them updated throughout the algorithm evolution) may instead be detrimental to the overall solution process. Another crucial aspect concerns where and when a model should be queried, that is whether performing fine or coarse-grained decisions and how often to update them (query frequency). The investigation of strategies to effectively guide a search process is an open and challenging topic that is becoming more and more relevant given the increasing capacity of exploiting large amounts of collected data.

Apart from tree search based resolution algorithms, a lot has been done in the area of metaheuristics with a special focus on fast objective function approximation (especially in population-based algorithms), initial solution generation, runtime operator selection (e.g. mutation in GA), restricting local search application to the best candidates. For works surveying the SOTA refer to [STÖ19; Cal+17].

Despite the challenges, a balanced integration may be beneficial for the resolution of the optimization problem. In fact, there are several strategic decisions that right now are made by following the designer's sensibility and past experience, e.g. when to switch between intensification and diversification, whether to accept worsening solutions, etc, or where full enumeration is still used.

Search Guidance as a transition system

In [ALW17], a trained learner h^*, defined over an (augmented) decision variable's space, is queried at each B&B node to estimate the strong branching score associated with fractional variables. Assuming a minimization MILP, this can be defined in our transition system as follows.
The combinatorial model is designed by an expert

$$\mathcal{M} = \langle \emptyset, \emptyset, \emptyset \rangle \underset{m}{\rightarrow} \langle X, f, g \rangle$$

and it is used as the starting point for the $(\underset{s}{\rightarrow}\underset{g}{\rightarrow}\underset{z}{\rightarrow})^*$ resolution process, $\mathcal{R} = \langle \{\mathcal{M}\}, \emptyset, \mathcal{M} \rangle$.
To fully specify the approach we define

- the model scoring function

$$\hat{z}(\mathcal{M}) = \hat{z}(\langle X, f, g \rangle) = \begin{cases} f(X), \text{ if } g \text{ are satisfied} \\ \infty, \text{ otherwise} \end{cases}$$

- the select scoring function

  $\hat{s}(M) = \{\mathcal{M}\}$ according to the default solver node selection strategy

- the generating function

  $$g(\{\mathcal{M}\}) = g(\langle X, f, g \rangle) = \cup_{cut \in branch(x^*)}(\langle X, f, g \rangle \underset{r}{\rightarrow} \langle X, f, g \cup cut \rangle)$$

  with the branching variable x^* selected according

  $$x^* = \arg \max_{x \in X \setminus X_{int}} \{h^*(\Phi(x))\}$$

  Other works, for instance [HDIE14], employ h^* for node selection, hence replacing the select scoring function $\hat{s}$.

Heuristic bounding

The bounding procedure is of paramount importance during the resolution of an optimization problem: the idea is to cut out regions of the feasible space that do not contain the optimal solution to the problem at hand. This can

have a massive impact on the computational performance of the resolution algorithm. Finding an explicit formulation for a bounding function, or designing an algorithm to compute it, is often impossible and bounding mechanisms rely either on historical solutions or relaxations of the current problem.

Moreover, defining a bounding function can be seen both as an aid to the resolution procedure, but also a tool to empirically evaluate the quality of an optimization algorithm on newly introduced problem instances for which no optimal solution is available and a compact and tight formulation is not available.

In this sense, bounding functions can be used to evaluate actions before actually performing them. As an example, in [BL+18] a neural network is used to estimate the improvement associated with variable cuts in order to select the most promising one (in accordance with the predicted bounds). Alternatively, a tree search process can be boosted by estimating a bound at each node via a predictor; in [HTT20] a neural network is used to prune branches in a heuristic tree search. This can dramatically speed up the solving process at the expense of optimality. In [Nai+20], authors exploit a learner for neural branching in a MIP problem; a trained deep network imitates a Full Strong Branching policy (here they use the ADMM as the target policy). The MIP parameters are embedded in a bipartite graph structure and fed to a Graph Convolutional Networks to extract a node embedding: this latter serves as input to a MLP that infers a bound on the current problem, imitating the way a primal heuristic works. In [FF19; Pra+18] ML is used to predict the optimal solution value of a problem instance, once the cost coefficients are known. Learning models can be also employed to predict constraint satisfaction, as done in [XKK18]. On the same line of research, in [Sel+18] a learner based on Neural Networks is exploited for predicting the satisfiability of SAT problems, using as a target a single bit (0 if false, 1 if true). The architecture of the network is based on a graph embedding, assuming each SAT problem to be formulated in Conjunctive Normal Form; the architecture maps literals and clauses to nodes, linked to each other to match the CNF of the input problem. Then a RNN is in charge of taking care of the Message Passing between nodes, at each iteration. The process is repeated until a general agreement between so-called 'voting' nodes is reached and the problem's output is clear. Although simple and not competitive with SOTA SAT solvers, the learner is able to have an accuracy of around 85% on SAT instances of the test set and moreover to retain a good accuracy also on instances of bigger size and coming from different problems (in fact, an advantage of the method is that any Boolean satisfiability problem

can be reduced to CNF).
In general, exploiting a ML model for bounding requires extra care: the training data set is required to be representative of the instance population, and the resolution process should pay close attention to eventual mispredictions. On one hand, when the bound is overestimated, it could lead to cutting feasible regions containing good (or optimal) solutions; in the case of underestimation, instead, the bound will be impossible to reach.
A technique to overcome these problems is presented in [SG20]; the authors introduce BION, a method for boundary estimation in constrained optimization problems: the idea is to learn, starting from historical solutions, to infer the value of the objective function. By enriching the instance features with additional hand-crafted ones, the method can be extended to different models and solvers, while the existence of a solution is guaranteed by introducing a *rollback* mechanism in charge of relaxing the stricter constraints.
A novel framework for ML-based bounding is proposed in [Cap+21b], where the authors combine Reinforcement Learning and Constraint Programming to boost the resolution process of a COP. The idea is to encode a COP within a Dynamic Programming framework by means of the Bellman equations, which allows to formulate the problem in both RL and CP environments. First, the agent is trained on randomly generated COP instances and then, during the B&B resolution algorithm, bounds are added to the CP problem based on the policy learned previously. This allows the author to solve complex COP problems like the TSP with Time Windows and the 4-Moments Portfolio Optimization. The approach can retain optimality and feasibility guarantees while reducing the computational time and is competitive with SOTA algorithms.

Heuristic Bounding as a transition system

Suppose we have an estimated bounding function (or procedure) h^*; there exist two main ways of exploiting it:

- *Global bounding*: predicting the optimal value associated to an instance I to support and ease the model resolution by pruning unpromising search trajectories

  1. The optimal solution value for instance I is estimated by using a trained machine learning model h^*

$$z^* = h^*(I)$$

2. A new constraint on the objective function f is added to the initial model
$$\langle X, f, g\rangle \underset{r}{\rightarrow} \langle X, f, g \cup \{f(X) < z^*\}\rangle$$
This new constraint is considered by the generating function g that will skip all models having an objective function greater than z^*.

- *Local bounding*: predicting whether to explore a given model state

1. When an optimization model $\mathcal{M}$ is generated by the generating function $\hat{g}$, the bounding estimator h^* is queried to predict whether to add $\mathcal{M}$ in the subsequent pool of open model states
$$\begin{aligned} g &= h^* \circ \hat{g} \\ \hat{g}(\{\mathcal{M}\}) &= \{\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_n\} \\ h^*(\{\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_n\}) &= \{\mathcal{M}'_1, \mathcal{M}'_2, \ldots, \mathcal{M}'_\ell\} \end{aligned}$$

Algorithm selection and parameters tuning

Machine learning can be employed to rank resolution algorithms from a portfolio based on their expected success rate on a specific instance, in order to select the best strategy given a task and its boundary conditions [Kot16] or to promptly adapt algorithm parameters to the solution space currently under evaluation [BB10]. With similar scope, in [Bal+18] the authors study algorithm configuration via machine learning in the tree search context. More precisely, they examine how to use machine learning to determine the optimal weighting of branching policies in B&B contexts. An offline classifier is employed in [BLZ18] to predict the most suited resolution approach to quadratic problems with the CPLEX solver. In this case, the classifier has to decide whether to linearize or not the quadratic part of the problem, based on the instance features.
Other works can be broadly classified in this category, although they do not strictly deal with algorithmic tuning: for instance, learning models can be used to provide hints to optimization problems. In [XQA21], the authors exploit a parameter vector to provide a COP with additional features, used to represent different aspects of the model to be solved, in particular: 1) the constraints to be used, 2) potential warm-start configuration of variables and 3) affine space where the solution is likely to lie. The method preserves the feasibility of the obtained solution, although can lead to sub-optimalities or slow resolution;

the idea is to learn recurrent features from historical solutions and use them to boost future resolutions.

2.3 Hybrid methodologies

The methods presented in this Section are strictly connected with ML, except for surrogate models. However, even if the scope of application is different, the process of constructing a surrogate model is loosely related to the training operation. For this reason, the schematic representation in terms of transition systems results very similar for all of the methods we will present.

2.3.1 Surrogate models

In Section 2.2.1 we have seen different techniques to tackle the scenario in which we need to devise an optimization problem but cannot model part of the process. If we have access to labeled data we can train a machine learning algorithm to emulate the real process. A similar situation can be found in black-box optimization: as the name suggests, we deal with unknown processes but have access to an oracle that can tell us the outcome for any given input. In other words, we cannot look inside the black box, but we can control what goes in and see what comes out. As a mere example consider simulation programs that may consist of several nested operations and can take hours to run, or quantities not having an analytical form as the results of numerical computations.
Problems of black-box optimization can rely on different techniques, namely heuristic methods, where queries to the oracle are guided by a heuristic algorithm, or Derivative-Free Optimization (DFO). This latter deals with the fact that, since we do not know the mathematical equations describing (part of) the process, it results impossible to use any method that employs gradient-based algorithms. An alternative solution is to use surrogate-based techniques to learn a simplified algebraic model, that is the surrogate, via interactions with the oracle. This is a modest but reliable shadow of the true underlying process and will be used as an approximation of the original one. The fundamental requirement is for such a model to be easy to optimize. More generally, the method applies whenever part of the process does not have a closed form expression, although is mostly suited for computationally expensive cases. Surrogate models were initially conceived for unconstrained continuous optimization

problems: constraints can be introduced either as restrictions on the domain of the surrogate model or penalizing undesired regions.

Although similar to the standard ML techniques, surrogate-based methods focus on the algebraic structure to deal with very low numbers of available samples. The grounding idea is to have a controlled model that is used for local search and iteratively refined during the process, on the basis of its performances. Different strategies are characterized by the way the data are sampled, i.e. the Design of Experiment (DOE), which requires a careful balancing between exploration of the domain region and computational tractability, and in the algebraic form of the surrogate model. This latter is usually constructed by summing up known functions, for instance, polynomials, Gaussian functions, Radial Basis Functions (RBFs), and so on.

A very good introduction to surrogate models can be found in [Que+05], while in [Vu+17] different methods are surveyed with a special focus on both the sampling of the domain space and the construction of algebraic models; finally, their application to black-box optimization problems is studied. In [CSM14], the authors devise a regularization approach to reduce the complexity of models constructed as a linear combination of non-linear basis functions (polynomials, transcendent functions, and so on). They resort to a model reduction analysis, also known as the "best subset problem" to find a subset of the original basis functions that minimize a given goodness-of-fit measure while reducing the number of basis functions used. Through mathematical reformulations, they can cast it as a MILP problem, which allows them to consider a large number (ca. 250) of basis components.

Surrogate Models as a transition system

We can depict the process of building and fitting (or training) a surrogate model as follows:

1. The functional form of the surrogate is defined, along with its loss function l (usually describing an error w.r.t. the target data)

$$\langle \emptyset, \emptyset, \emptyset \rangle \underset{m}{\rightarrow} \langle \theta, \ell, \emptyset \rangle = \mathcal{M}$$

2. The initial resolution state is then represented by

$$\langle \mathcal{M}, C = \emptyset, b = \mathcal{M} \rangle$$

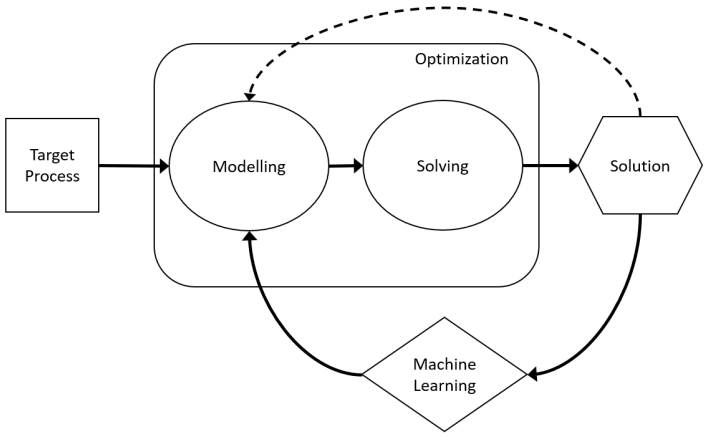

Figure 2.4: Machine learning is used to boost the modeling phase of an optimization problem in an End-to-End scheme.

3. A set of input examples are drawn from the domain, $x_i \in \mathcal{D}$ and the surrogate model is trained by means of a (sequence of) transitions $(\underset{s}{\rightarrow}\underset{g}{\rightarrow}\underset{z}{\rightarrow})^N$ as done with learning models.

4. The model is updated by sampling new points in correspondence to the most promising areas. This is very similar to the description of supervised learning as a transition system, given in Section 1.3.1, but the data sampling process implies an additional modification to the objective function of $\mathcal{M}$.

2.3.2 Task-Based Learning

The methods seen in Section 2.2.1 treat learning and optimization as completely separated processes: first, we train the machine learning model on a given task, then we embed it into the optimization problem and solve the latter, which results in a two-stage process.
However, keeping two phases completely independent of each other is not always a good practice, for the learning and optimization phase could be guided by different loss functions, which eventually leads to poor overall performances. A solution that has recently been proposed is that of pulling together learning and optimization in an end-to-end scheme, as depicted in Figure 2.4: as before, the learner is used to model unknown relations or parameters on which the op-

timization problem depends. However, and here lies the novelty, the learner is not pre-trained on a given dataset and then embedded in the optimization, but is trained *on-the-fly* during the resolution process, computing the loss on the optimization results it induces: this is done by back-propagating the loss of the objective function up to the original source, that is the learning model. The approach is radically different from the usual two-stage one and is grounded on the intuition that, when used in a larger process, we are not seeking the most accurate learning models, but those producing the overall best solutions, evaluated with respect to the objective function of the decision problem. That is, we evaluate the models on the final objective function of the process, which can correspond to a sub-optimal learner configuration.

The methodology is usually referred to as *task-based learning* or *decision-focused learning* and a seminal concept can be found in [Ben97]. A first application of the method in the context of stochastic programming can be found in [DAK17], in [WDT19] it is extended to combinatorial problems such as linear programming and submodular maximization; both works make use of the KKT condition to differentiate through the argmin operator of the prescriptive problem (provided the constraints are differentiable!). In [Fer+19] a framework for general MIP problems is proposed: the idea is to deal with integrality constraints by generating cutting planes - as in the usual branch-and-bound algorithm - until the corresponding LP relaxation is integer; then the algorithm proposed in [WDT19] is applied. Another approach for task-based learning is proposed in [EG17], where the predictor is in charge of producing the costs associated with a linear objective function: the authors introduce a convex surrogate loss function to overcome the problem of a discontinuous loss function, which allows them to tackle any convex or MIP problem with a linear objective function. By explicitly computing the gradient of such loss function, they can employ a gradient-based method for solving the whole problem.

Note that task-based learning is usually done by hand-crafting a problem specific surrogate loss function that is convex and differentiable. This recalls what is done in optimization with heuristic approaches and shares the same problems: it might require an expert hand a considerable amount of effort to craft the proper surrogate function. In [Che+19] the authors tackle the problem of crafting a surrogate task-based loss function and propose an algorithm to learn it from the true task loss, instead of modeling it. [Pau+21] extend the idea of embedding COPs within a neural architecture to Integer Linear Problems: the backward pass is computed by differentiating through a proper proxy function that embeds both the integrality constraints and the problem specifics. This

makes it possible to extract ILP parameters, such as costs and constraints, as higher-level features from data, while using the network to directly output the solution to the combinatorial problem. The methodology is tested on synthetic datasets, a knapsack problem, and a keypoint matching problem, proving to be superior to any method relying on LP relaxation.

Task-Based Learning as a transition system

In task-based learning, we aim at training a machine learning model that will eventually be used to define some components of an optimization problem. With respect to standard "Predict, then Optimize" methods, where training and optimization phases are independent of each other, these approaches are characterized by a training process that now takes into account the solution of the model induced by its current parameters.

1. First a model state is defined by initializing the learner's parameters

$$\langle \emptyset, \emptyset, \emptyset \rangle \underset{m}{\rightarrow} \langle \theta, \ell, \emptyset \rangle = \mathcal{M}$$

2. The initial resolution state is defined

$$\langle M = \mathcal{M}, C = \emptyset, b = \mathcal{M} \rangle$$

3. The learner parameters are optimized by the $(\underset{s}{\rightarrow}\underset{g}{\rightarrow}\underset{z}{\rightarrow})^N$ sequence of transitions in which

$$\begin{aligned} \hat{s}(\mathcal{M}) &= \mathcal{M} \\ \hat{g}(\mathcal{M}) &= \mathcal{M}' \\ \hat{z}(\mathcal{M}) &= \min(-l(\theta), -l(\theta')) \end{aligned}$$

   where $M' = \langle \theta', \ell, \emptyset \rangle$ is a minimization problem and θ' is defined by a gradient descent application dependant on the resolution of an optimization problem $P = \langle X, f(X, \theta), g(X, \theta) \rangle$ by means of a $(\underset{s}{\rightarrow}\underset{g}{\rightarrow}\underset{z}{\rightarrow})^*$ inner resolution process starting from $\langle M = P, C = \emptyset, b = \mathcal{P} \rangle$

4. The model produces an instance of the optimization problem and its solution used to refine the learner's parameters

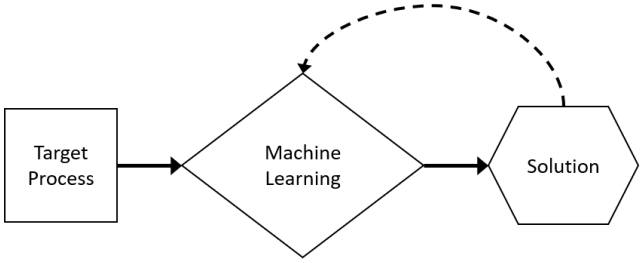

Figure 2.5: Machine learning is used to entirely replace the usual optimization scheme and directly output the solution to the COP.

2.3.3 End-To-End Learning

When a combinatorial problem is NP-hard, it often becomes prohibitive to solve it with exact methods: the resolution time can be impractical for the problem to be of any use, therefore it is often handled via heuristic search procedures.

Many authors have been studying if, and how, the heuristic search can be replaced with automatic learning methods: the underlying idea is to leave to the learner the task of devising an effective search procedure and then use it to output an entire solution, in an end-to-end fashion, as sketched in Figure 2.5. This has a double advantage over handcrafted heuristics: the agent might be able to capture the problem's structure more efficiently than an expert and it can adapt when the problem's boundary conditions change. In fact, one of the recurrent problems with heuristic methods is their scarcity of generalization, which makes them very problem-specific; this can be partly mitigated by means of meta-heuristics and hyper-heuristics algorithms, however, it results hard to balance the trade-off between generalization and accuracy of the methods.

When it comes to *End-To-End* Learning, the most preferred paradigm is that of Reinforcement Learning. There is an important aspect behind this precise choice: supervised learning needs labeled data to be trained. In the case of a combinatorial problem, each example of the training data set corresponds to an optimal solution; however, it could happen that we cannot afford to compute the exact solutions to NP-hard problems, or it is too expensive to retrieve enough solutions for a deep, highly parametric learning model to be properly trained. Thus, the paradigm of reinforcement learning comes in handful: we

just need to formulate a proper reward function to rank each of the agent's possible actions and a policy to choose among them. The former is straightforward in our case, as we can simply use the objective function of the problem. Furthermore, once the reinforcement learning environment is set up, we can allow the learner to explore all the configuration space, rather than feeding it with a (small and) finite set of solutions, as in the case of supervised learning. Another advantage of reinforcement learning over supervised learning is that we are able to devise novel, and possibly competitive, solutions rather than replicating the results of known algorithms.

A seminal work was first proposed in [Bel+16], where the authors face the Travelling Salesman Problem (TSP) in two dimensions. The approach is grounded on Pointer Networks [VFJ15], a neural architecture introduced to deal with outputs of variable size and used for combinatorial problems in a supervised learning setting. In pointer networks, sequence modeling is blended with an attention mechanism; the latter is in charge of selecting the next element of the output sequence from the input one and can thus be applied to problems with discrete variables and different input sizes. In [Bel+16], the authors rely on learning a policy with the `REINFORCE` algorithm to choose the order of points to visit, such that the overall tour length is minimal. The policy is learned through an actor-critic dual network, where the latter is used in the `REINFORCE` algorithm to estimate the tour length for a given policy. The results improved with respect to the supervised learning baseline [VFJ15] and achieved solutions close to optimality for instance with up to 100 nodes, with performances decaying with the increasing of the number of nodes.
In [Naz+18], the authors replace the encoder used in Pointer Network, a Recurrent Neural Network, to get a sequence-invariant embedding of the inputs and tackle the Vehicle Routing Problem with split deliveries and a stochastic variant. The same approach is investigated in [KVHW18], with the authors improving the learning algorithm and network architecture: the encoder takes care of embedding the input graph by means of a Multi-Head Attention mechanism and the decoder is fed with an additional context embedding to represent the different instance features. They extend it to two variants of the Vehicle Routing Problem (VRP) and can improve against previous results. A very similar architecture is used in [Deu+18] and applied to TSP instances.

A different approach, specifically intended for combinatorial problems defined over graphs, is proposed in [Dai+17]. They design an algorithm that combines reinforcement learning and graph embedding to learn a greedy policy

that iteratively builds a feasible solution. In contrast to sequence-to-sequence approaches [VFJ15], the solution will be constructed by adding one element at a time by greedy selecting the most promising node to add to the current solution. The graph embedding is represented by a non-linear propagation between neighboring sites, entrusted to a neural network named `structure2vec` [DDS16]. The method is applied to a range of combinatorial problems over graphs such as Minimum Vertex Cover, Maximum Cut, and TSP and outperforms standard heuristic and sequence-to-sequence approaches.

These works require in general thorough crafting but have the potential of scaling to different instances and, even more interesting, to different instance sizes.

Following [Dai+17], many works focused on Graph Neural Networks [Sca+08] to deal with COP data: as the name suggests, this architecture is explicitly made to deal with graph-like inputs. the idea is to associate to each node of the graph a vectorial representation and, by means of neural connections, represent the graph topology by stacking network layers. A GNN is then made of several layers and each of them aggregates the local information of a node's neighbors. Here we list just a few of the many recent works that explored this novel area, while we redirect the interested reader to the good survey [Cap+21a] that specifically reviews the use of GNN in COP and how the former can be employed to aid the resolution, but also explore algorithmic reasoning with GNNs, i.e. introducing in the network's structure or learning paradigm concepts from classical algorithms to bias the learning process towards known structures. One of the initial works in the field of reasoning with GNN is [Xu+19], showing how GNNs architectures align with Dynamic Programming, a language that can represent most of the combinatorial algorithms. The algorithmic alignment seems an important feature for the learned reasoners to work well in extrapolation regime.

[Ma+19] introduces Graph Pointer Networks, which inherits the architecture of Pointer Networks with an additional graph embedding layer on the input. The idea is to feed all the node coordinates of the problem to a Graph Neural Network, instead of a single point, to produce a *vector context* that contains much richer information; this extends the original Pointer Network structure to comprehend the full graph feature. In addition to this structure, the authors use a hierarchical methodology to gradually learn the final probability distribution over the nodes; the idea is to define different learning policies, with the lowest providing constraint satisfaction, and the highest refining the solution

to minimize the true original loss function: the original hard constraints are relaxed ad added as penalty terms in the objective function. By combining the GPN architecture with reinforcement learning, the authors are able to solve the TSP with Time Windows and also scale to instances with bigger sizes at inference time.

In [JLB19], the TSP problem is tackled by means of a non-autoregressive approach based on a neural network that outputs a heatmap over the edges to be used in the solution of the COP. This latter is combined with a heuristic search procedure (greedy or beam) to produce a valid solution. The learner architecture is based on a Graph Convolutional Network, which is fed with the encoded graph instance and has a MLP on the output, to convert the edge/node embeddings into the corresponding probability. The loss function is constituted by a (weighted) binary cross-entropy, using as a reference the exact solution to the TSP problem and having non-zero values in correspondence with the used edges.

In [Dro+20], the authors first define a unified framework for graph COP, by means of line graphs; the latter is the dual of a graph, obtained by switching its vertices with edges and vice versa. This allows using the same learning paradigm to deal with different COPs, i.e. the VRP, the Minimum Spanning Tree (MST), and Shortest Path (SSP). The learner is constituted by a Graph Attention Network (see [Vel+17]) and trained via Reinforcement Learning in a Single Player Games fashion (searching the tree representing all possible solutions to the problem), in the classic encoder-decoder scheme, with an attention-based decoding scheme. In this way, the network provides probabilities over the embedded graph nodes, which are then passed to a greedy selection policy. The results show competitive performance with SOTA heuristic algorithms as well as an overall improvement with respect to other End-To-End learning schemes, with the method being able to generalize to different COPs and also to different instance scales.

Other works include [Lem+19], where a GNN is employed to solve the Graph Coloring Problem, in [YP19] a RL scheme is employed to learn a SAT solver heuristic from scratch with curriculum learning (training on instances of increasing difficulty).

We can depict the general structure of End-To-End learning methods as shown in Figure 2.6 and distinguish them on the basis of the different sub-tasks that are usually employed.

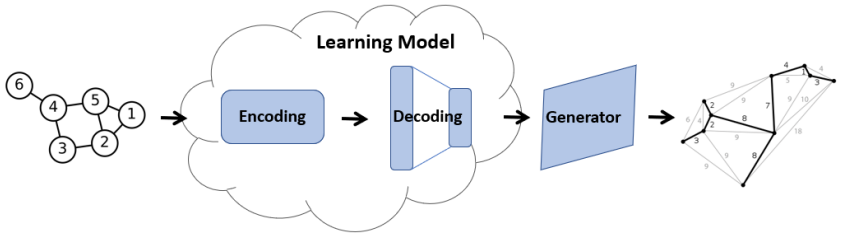

Figure 2.6: General architecture of the End-To-End learning schemes: a graph instance is passed to an encoder to obtain a suitable graph embedding. A decoding scheme acts on the embedding and is coupled to a generating algorithm to output the complete solution.

End-To-End Learning as a transition system

Reinforcement learning can be employed to learn the best policy to solve an optimization problem, once the objective function is given. The policy is then used to construct the solution associated with any instance and acts, in fact, as a constructive algorithm. In the examined works, the points of contact between machine learning and optimization can be summarized in two key concepts:

- Encoding/decoding of the combinatorial structure of the solution: this can be done by means of ad-hoc embedding algorithms (structure2vec, graph embedding) or learning models (pointer networks, attention models)
- The reward function takes into account the true objective function of the combinatorial problem to identify the best policy; this latter is used to compute the solution associated with any instance and its corresponding loss $\tilde{\ell}$.

In the transition system framework, these methods can be described in a very similar way to *task-based learning*, with an initial model state

$$\langle \emptyset, \emptyset, \emptyset \rangle \underset{m}{\rightarrow} \left\langle \theta, \tilde{\ell}, \emptyset \right\rangle = \mathcal{M}$$

in which the loss function is $\tilde{\ell}$ and can describe either supervised or reinforcement learning algorithms. For instance, in the case of [KVHW18], the learner

is composed of a multi-head attention encoder coupled with a decoder built upon recurrent neural networks. The model's parameters are updated via the REINFORCE algorithm, which provides a gradient estimator given a baseline $b(s)$, given an instance s:

$$\nabla \ell(\theta|s) = \mathbb{E}_{p_\theta} \left[(L(\pi) - b(s)) \nabla \log p_\theta(\pi|s) \right]$$

and the gradient update is

$$\theta' = \mathrm{Adam}(\theta, \nabla \ell)$$

Category	Training	Method	Encoding	Decoding	Generator	Applications
AR	Supervised	[VFJ15]	LSTM	LSTM + Attention	Greedy	TSP
	RL	[Bel+16]	LSTM	LSTM + Attention	Greedy	TSP, Knapsack
	RL	[Dai+17]	Structure2Vec	Q-learning	Greedy	TSP, MaxCut, MVC
	RL	[Naz+18]	D-dim embedding	LSTM + Attention	Greedy	VRP, CVRP
	RL	[KVHW18]	Multi-Head-Attention	LSTM + Attention	Greedy	TSP, OP, PCTSP
	RL	[Deu+18]	Multi-Head-Attention	LSTM + Attention	Greedy	TSP
	RL	[Ma+19]	GNN + Attention	LSTM + Attention	Hierarchical RL	TSP, TSPTW
	RL	[Dro+20]	GNN + Attention	RNN + Attention	Greedy	TSP, MST, SSP
Non-AR	Supervised	[JLB19]	GCN	MLP	Beam Search	TSP
Non-AR	Supervised	[Now+17]	GNN	-	Beam Search	QAP

Table 2.2: End-To-End methods classified by category (AutoRegressive or not), training algorithm and components, as sketched in Figure 2.6

2.4 Common Themes

The methods presented in this Chapter span a diversity of situations, both for the type of learning algorithm employed and the interaction with the resolution process. However, we can identify some common traits that permeate the field of integration between Machine Learning and Constrained Optimization.

Modelling Hints Learning models can be used to extract inner representations that can be used to gain insights on the problem structure and characterize novel heuristics. This is especially true for methods employing RL, for the policy optimization process is agnostic of the problem resolution methods, but guided solely by the reward function. The learned heuristic policy can thus exploit structures of the problem that were not known before. On the

other hand, algorithms based on Imitation Learning are implicitly aware of the resolution methods, for these are employed to generate the solution on which the learners are trained.

Generalization Supervised learning models are trained on a subset of the configuration space corresponding to the training set $\mathcal{D}$; this may cause troubles when the learner is integrated within a COP, for the latter usually entails a complete exploration of the feasible space. For this reason, devising a proper test set to evaluate the performance of the learned heuristic requires extra carefulness and yet does not guarantee the same performances will be preserved during the optimization process.

Optimality The feasibility of solutions generated by learning algorithms is usually impossible to guarantee from a theoretical point of view. This is especially true for methods employing ML to produce complete solutions for the COPs. The drawback can be partially mitigated, or completely avoided, when the output of the learner is employed within the resolution process but the latter is carried on by a solver.

Scaling Learning models show high computational demand for training, both in terms of computational power and resources needed to generate the instances. However, their generalization capability may not compensate for the effort. The novel methods (i.e. RL for COPs) that have appeared brought unprecedented results, that were inconceivable before; however, their extension to more complex problems, larger instance sizes, or even real-world scenarios already seem out of reach. For example, consider the results on the resolution of TSP: they have been successfully extended to more complex variations of the problem, such as the VRP, OP, and CVRP. However, graph sizes of a few thousand nodes are already very challenging for such methods, while being standard if not small size for real-world instances (and we are disregarding the constraint aspect).

2.5 Conclusions

In this Chapter we reviewed several methods in the field of integrating learning methods in COPs; although research is at a very early stage, promising results and expected potentialities are driving an intense research work. This survey

is not exhaustive and surely misses many works (also due to the ever-increasing number of publications in the sector), however, tries to give a classification of the methods according to the integration aim, together with a methodological description of how such integration is carried out and affects either the modeling or the resolution phase of an optimization process.

The research direction that has received more attention in the last years is probably the one we classified as *hybrid algorithms*; these novel methods disrupt the traditional COP distinction between modeling and solving phases but instead combine them together. In particular, methods exploiting Reinforcement Learning are considered very well-suited to deal with COP because of the easy identification of the policy reward with the problem objective function and the possibility to fully exploit computational resources in an end-to-end training scheme, rather than restricting to an offline generated data set. As seen before, this comes at the cost of potentially complex hand-crafted representations of feasible solutions. On the other hand, algorithms based on Imitation Learning, despite being the first to appear, present several limitations that have to be taken into account, for instance, their generalization capability.

A further promising direction is that of using ML to replace heuristic choices within the resolution process of COPs. This doesn't undermine the optimality and feasibility guarantees of the resolution process while having the potential to boost the computational performance and devise more efficient algorithms.

Chapter 3

Moving Targets

This chapter introduces Moving Targets, an algorithm conceived to inject constraints into Machine Learning models by coupling the training process together with a Constraint Optimization solver. This latter acts in fact as a validator of the learner and evaluates it on the basis of its predictions.
The training phase of the predictor is guided by the validator, which adjusts the targets against which the predictor is trained, thus dynamically driving it towards the feasible region but retaining the two problems independent of each other. Unfortunately, the bi-level optimization setup lacks theoretical convergence guarantees, except for simple scenarios. Thorough experiments will try to compensate for this shortcoming, together with a comparison with well-known methods; this chapter contains results published in [DLM21]. Our code and results are publicly available at `github.com/fabdet/moving-targets`.

3.1 Motivation

The possibility of dealing with constraints in Machine Learning (ML) would add enormous value to the field and has the potential to address outstanding issues in data-driven AI methods. Constraints can be seen as mathematical representations of external knowledge that can be used to boost the performance of a ML model. This, coupled with the expressiveness and flexibility that constraints are capable of, opens the way to a more controlled, robust, and user-tailored AI.

For example, constraints representing physical laws can be employed to correct misbehavior but also to improve the generalization capability of the learned model by biasing its behavior in extrapolation regime; they can encode known patterns among the data (e.g. excluded classes) or relational information between multiple examples. Furthermore, constraints can be used to guarantee the satisfaction of required properties, such as fairness, safety measures, lawfulness, and so on. They can even be used to extract symbolic information from data.

As a mere example, consider the case of a physical system that is partially known (e.g. we know the physical laws between some variables): a data-driven model could be able to correctly interpolate the relation between the observed data, but it is very likely that will fail as we move further from the training set distribution. However, since we know how some of the variables are related, we can force the model to be consistent with the physical laws by superimposing these latter as constraints.

However, the high flexibility that can be expressed via constraints often comes with a high cost: this external knowledge has to be injected into learning techniques. The vast majority of approaches to deal with it in ML make assumptions that either restrict the type of suitable constraints, usually differentiability and no relational information, the type of models, for instance, Decision Trees or gradient-based learning models, and often force modification in the employed training schemes (e.g. specialized loss terms).

In this Chapter, we propose a decomposition-based method, that we call Moving Targets, to enable supervised learning with constraints. In our bi-level optimization scheme, a master step "teaches" constraint satisfaction to a learning algorithm by iteratively adjusting the sample labels. This indirect communication guarantees that master and learner have no direct knowledge of each other, which provides several advantages: 1) any ML method can be used for the learner, off the shelf, and without further modifications; 2) the master can be defined via techniques such as Mathematical or Constraint Programming, to support discrete values or non-differentiable constraints, but can also be replaced by heuristic algorithms. Our method is well suited to deal with relational constraints over large populations (e.g. fairness indicators), as the experiments show. Moving Targets subsumes the few existing techniques – such as the one by [KC09] – capable of offering the same degree of versatility.

As will be clear in the subsequent sections, our approach prioritizes constraint satisfaction over accuracy. For this reason, it is not well suited to deal with fuzzy information or unreliable constraints. Moreover, due to the very open

setting of MOVING TARGETS, convergence properties are very hard to be established.
However, due to its combination of simplicity, generality, and the observed empirical performance, MOVING TARGETS can represent a valuable addition to the arsenal of techniques for dealing with constraints in Machine Learning.

3.2 Problem Statement

We restrict to the case of *supervised learning*, where labeled data is available and the training process consists of minimizing the empirical loss function over the training data set (see Section 1.2.2). In the presence of constraints, we can represent the latter via a feasible set C, such that the original optimization problem can be formulated as follows:

$$\arg\min_{\theta}\{L(y, y^*) \mid y = f(X;\theta), y \in C\} \tag{3.1}$$

where f represents the ML model and θ its parameter vector. With some abuse of notation, we refer as $f(X;\theta)$ to the vector of predictions for the examples in the training set X. Since the model input at training time is known, constraints can be represented as a feasible set C for the sole predictions y. Let $L(y, y^*)$ be the loss function, where y is the prediction vector and y^* is the label vector. We make the (non restrictive) assumption that *the loss is a pre-metric* – i.e. $L(y, y^*) \geq 0$ and $L(y, y^*) = 0$ iff $y = y^*$. Examples of common loss functions can be found in Table 3.1.
The problem described by Equation (3.1) is generally difficult to tackle, except for simple cases. An exact resolution would require the use of methods from Constrained Optimization (Section 1.2.1), however, this has poor outcomes because 1) the learning function could be non-linear, which makes the optimization problem really hard and 2) we need to model this function for each example of the training set, that can result in many thousands of constraints. Both factors contribute to yield the constrained learning problem intractable with standard techniques.

3.3 The Algorithm

Our objective is to find a solution to Equation (3.1), which boils down to finding the best parameters θ of a ML model. We acknowledge that any

Loss Function	Expression	Target Space
Mean Squared Error	$\frac{1}{m}\|y - y^*\|_2^2$	$\mathbb{R}^m$
Mean Absolute Error	$\frac{1}{m}\|y - y^*\|$	$\mathbb{R}^m$
Hamming Distance	$\frac{1}{m}\sum_{i=1}^{m} \mathrm{I}[y_i \neq y_j^*]$	$\{1..c\}^m$
Cross Entropy	$\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{c} y_{ij}^* \log y_{ij}$	$[0,1]^m$

Table 3.1: Notable loss functions for the Supervised Learning problem (m is the number of examples and c the number of classes)

constrained learning problem must trade prediction mistakes for a better level of constraint satisfaction, for the feasible region C will inevitably affect the solution space. We attempt to control this process by carefully selecting which mistakes should be made. This is similar to [KC09; KC12; LRT11], but: 1) we consider generic constraints rather than focusing on fairness, and 2) we rely on an iterative process (which alternates "master" and "learner" steps) to improve the results.

The problem described by Equation (3.1) can be rewritten without loss of generality by introducing a second set B corresponding to the bias of the learning model. This leads to a formulation that is entirely expressed in label space:

$$\arg\min_{y}\{L(y, y^*) \mid y \in B \cap C\} \tag{3.2}$$

where $B = \{y \mid \exists\theta, y = f(X;\theta)\}$.

The Moving Targets method is described in Algorithm 1, and starts with a learner step w.r.t. the original label vector y^*, that we term pretraining. Each learner step, given a label vector as input, solves approximately or exactly the following problem:

$$l(z) = \arg\min_{y}\{L(y, z) \mid y \in B\} \tag{3.3}$$

Algorithm 1 MOVING TARGETS

input label vector y^*
scalar parameters α, β, n

1: $y^1 = l(y^*)$ ▷ pre-training (Equation (3.2))
2: **for** $= 1..n$ **do**
3: **if** $y^k \notin C$ **then**
4: $z^k = m_\alpha(y^k)$ ▷ infeasible master step Equation (3.4))
5: **else**
6: $z^k = m_\beta(y^k)$ ▷ feasible master step Equation (3.5))
7: **end if**
8: $y^{k+1} = l(z^k)$ ▷ learner step (Equation (3.2))
9: **end for**

Note that this is a traditional unconstrained learning problem since B simply represents the model and/or algorithm bias. The result of the first learner step gives an initial vector of predictions y^1. In other words, the vector y^1 is the result of training the ML model f on the data set X.
Next comes the master step to adjust the label vector: this can take two forms, depending on the current predictions.

- *In case of infeasibility*, i.e. $y^k \notin C$, we solve:

$$m_\alpha(y) = \arg\min_z \left\{ L(z, y^*) + \frac{1}{\alpha} L(z, y) \mid z \in C \right\} \tag{3.4}$$

  Intuitively, we try to find a feasible label vector z that is close (as measured by the loss function) to both the original labels y^* and the current prediction y. The parameter $\alpha \in (0, \infty)$ controls which of the two extremes should be preferred.

- *In case of feasibility*, i.e. $y^k \in C$, we instead solve:

$$m_\beta(y) = \arg\min_z \{ L(z, y^*) \mid L(z, y) \leq \beta, z \in C \} \tag{3.5}$$

  i.e. we look for a feasible label vector z that is not too far from the current predictions (e.g. in the ball defined by $L(z, y) \leq \beta$) and closer (in terms of loss) to the true labels y^*. When the current prediction vector satisfies the constraints, we seek an accuracy improvement within the feasible region C.

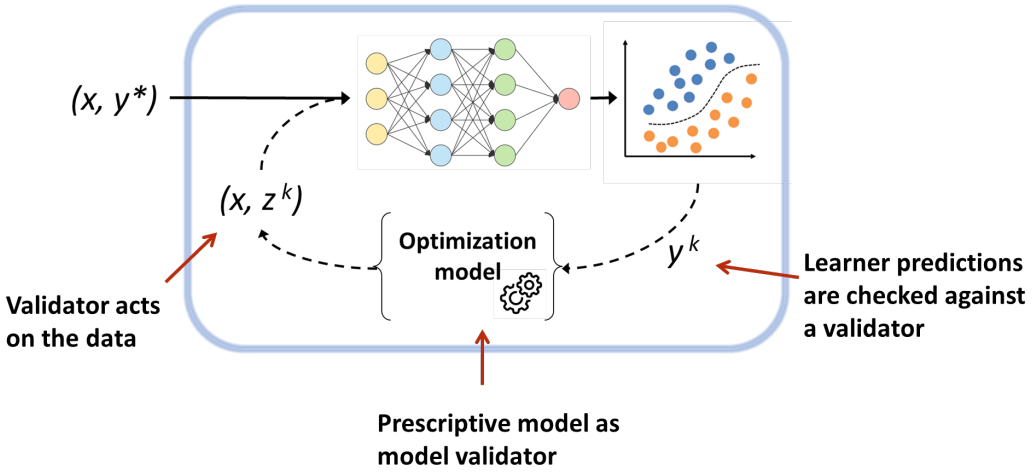

Figure 3.1: Sketch of the Moving Targets process: the prescriptive model acts as a validator of the current learner predictions and acts on the data by moving the target labels. These latter are then fed again to the learner until convergence or a stopping criterion is met.

After the master step, we proceed to a novel learner step trying to reach the adjusted labels; the new predictions will be adjusted at the next iteration, and so on. In the case of convergence, the predictions y^k and the adjusted labels z^k become stationary (but not necessarily identical). An example run, for a Mean Squared Error loss and convex constraints and bias, is in Figure 3.2, while the process behind Moving Targets is sketched in Figure 3.1.

3.4 Analysis and Convergence

A crucial aspect of any algorithm is given by its convergence behavior: in theory, we'd require any algorithm to have solid theoretical convergence guarantees, however, a thorough demonstration often becomes impractical. In the case of Moving Targets major difficulties arise because of the bias set B, which is often non-convex, and the discrete domain when it comes to classification problems. In this section, we will first review the main properties of the algorithm and then move to some important yet qualitative observations about its convergence features.

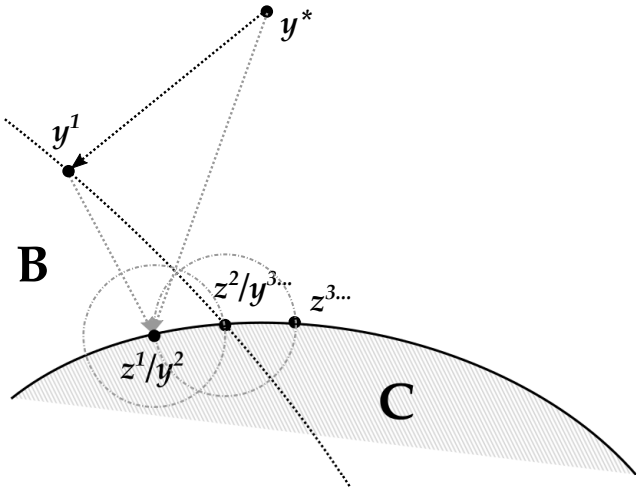

Figure 3.2: Example run of the algorithm: the pre-training step from y^* to y^1 is followed by a (infeasible) master step m_α from y^1 to z^1; then a learner iteration brings the predictor from y^1 to y^2 and is followed by a (feasible) master step m_β, from y^2 to z^2. The process continues until convergence is achieved, or a maximum number of iterations is reached.

3.4.1 Properties

The learner is not directly affected by the constraints, as expressed in Equation (3.3), thus enabling the use of arbitrary ML approaches without any modification. In the same way, the master problem does not depend on the structure of the ML model but uniquely on its predictions, often leading to clean structures that are easier to solve. For this reason, the master step can be addressed via any suitable solver, so that discrete variables and non-differentiable constraints can be tackled via (e.g.) Mathematical Programming, Constraint Programming, or SAT Modulo Theories. Notice that, depending on the constraints, the loss functions, and the label space (e.g. numeric vs discrete) the master problems may be NP-hard. Even in this case, their clean structure may allow for exact solutions for data sets of practical size. Moreover, for separable loss functions (e.g. all those from Table 3.1), the master problems can be defined over only the constrained examples, with a possibly

significant size reduction. When scalability becomes a concern, the master step can be solved in an approximate fashion.

3.4.2 Convergence

Due to its open nature and minimal assumptions, establishing the convergence properties of our method is hard. Here we provide some considerations and connect the approach to existing algorithms. First, we make the simplifying assumption that the learner problem from Equation (3.3) can be solved exactly. This holds for a few cases (e.g. convex ML models trained to close optimality), but in general the assumption will not be strictly satisfied. Observe that solving Equation (3.3) to optimality is usually undesired from a ML perspective, for it could lead to overfitting on the training data, while it is generally preferred to keep the model less biased to increase its generalization capability over unseen data. Observe also that Equation (3.2) simply corresponds to the Best Approximation Problem (BAP), which involves finding a point in the intersection of two sets (in our case the bias set B and the constraint set C) that is as close as possible to a reference point (the true label y^*). For closed convex sets in Euclidean spaces, the BAP can be solved optimally via Douglas-Rachford splits or other methods relying on projection operators, as described in [AC18]. Moreover, notice that our learner, equipped with a loss function that is a pre-metric, corresponds to a projection operator on B; then it would seem reasonable to couple the learning step with a projection operator onto the feasible set C to retrieve the aforementioned methods together with their convergence properties. Unfortunately, we cannot reliably assume convexity of the bias set B and we do not work solely in continuous space; in a discrete space, the Douglas-Rachford splits may lead to "label" vectors that are meaningless for the learner (in other words, they do not belong to the original discrete label space).

We, therefore, chose a heuristic design that may appear less elegant in the form and doesn't come with convergence guarantees but results less sensitive to which assumptions are valid, therefore more suitable for our general-purpose approach. In particular:

1. in the m_α step, we use a modification of a suboptimal BAP algorithm to find a *feasible* prediction vector: it is suboptimal because the training process can be carried out with heuristic methods, and modified for it

doesn't directly point to the projection of the original labels y^* onto the intersection $B \cap C$;

2. in m_β we apply a modified proximal operator to improve its distance (in terms of loss) w.r.t. the original labels y^*.

For reasonable α values, our m_α steps balances the distance (loss) from both the predictions y^k and the targets y^*. Convergence, in this case, is an open question, but especially in a non-convex or discrete setting (where multiple projections may exist), this modification helps escape local minima and accelerate progress. The behavior for a few values of α is depicted in Figure 3.3 for continuous labels and MSE loss.
When a feasible prediction vector is obtained, our method switches to the m_β step; we then search for a point in C that is closer to the true labels, but also not too far from the predictions. This is related to the Proximal Gradient method, discussed e.g. in [PB+14], but we limit the distance via a constraint rather than a squared norm, and we search in a ball rather than on a line. As in the proximal gradient, a too large search radius prevents convergence: for $\beta \to \infty$ the m_β step always returns the same adjusted labels, corresponding to the projection of y^* on C set.

3.4.3 Comparison with other methods

We compare our method to other algorithms with known convergence properties and sketch alternative methods to constrained ML.

- **Alternating Projection**: in the limit $\alpha \to 0$ the infeasible master problem $m_\alpha(y^k)$ becomes

$$z^k = \arg\min_z \{L(z, y^k) : z \in C\}$$

which is exactly a projection of y^k onto C, for the loss function is a pre-metric. Recall that the learning step, when solved to optimality, corresponds to either a projection of the target labels z^k onto the ML bias set B, when $z^k \notin B$, or the identity operator. In the first case, this is very similar to the Alternating Projection (AP) method [BD+03], where, as the name suggests, we alternate projection on two sets. The grounding idea is that at each projection we get closer to the point of minimal distance between the two sets. The method is proved to converge

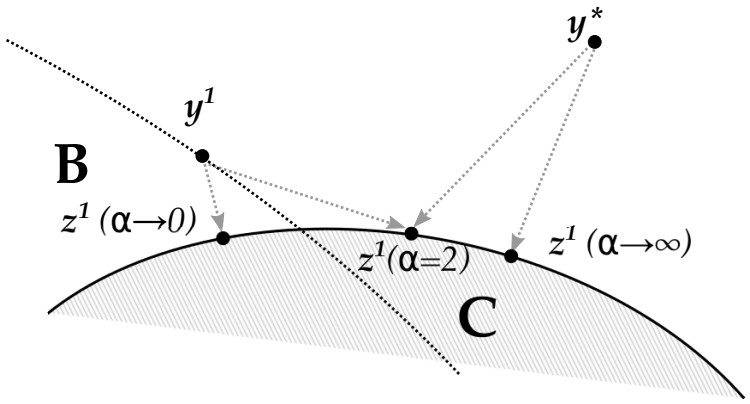

Figure 3.3: Effect of α in the infeasible master step when the target labels are continuous and the loss function is the MSE: for $\alpha \to 0$, the infeasible master problem m_α becomes a projection of the current predictor output y^k onto the feasible region. Conversely, when $\alpha \to \infty$, m_α becomes a projection of the true target y^* onto the feasible region C.

if the two sets are convex (also in the case their intersection is null). To compare with the AP method, we pick our sets to be the feasible region C and the model's bias region B: in most cases, C is convex but B does not, as is the case with neural networks. The AP algorithm is sketched in Figure 3.4.

- **Preprocessing**: in the limit $\alpha \to \infty$ we obtain essentially a pre-processing method as the one introduced in [KC09]: m_α becomes a projection of the true labels y^* on C, and subsequent iterations have no further effect; convergence to a feasible point is achieved only if the pre-processed labels are in B, which may not be the case (e.g. a quadratic distribution for a linear model).

- **Proximal Point**: the Proximal point (PP) method belongs to the family of proximal algorithms, described in [PB+14], and generalizes the concept of projection. A proximal point is defined as

$$z = \text{prox}_{\lambda f}(x) = \arg\min_{\xi} \left\{ f(\xi) + \frac{1}{2\lambda} ||x - \xi||_2^2 \right\}$$

where the parameter λ controls the cost to pay to move from x and f

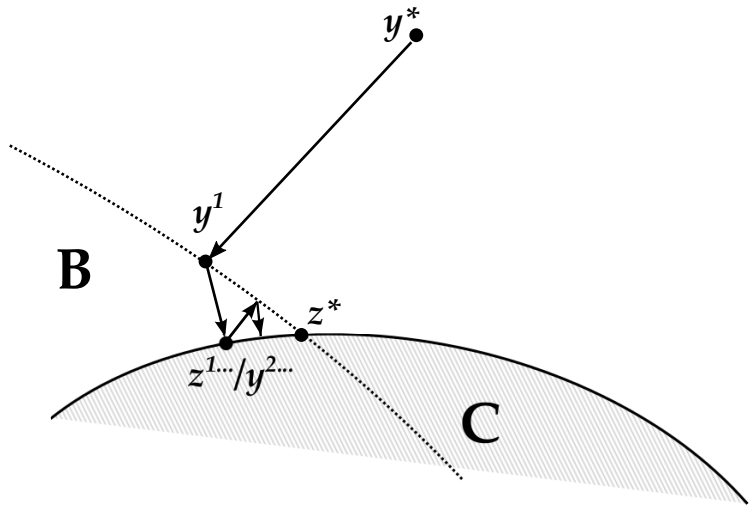

Figure 3.4: The behavior of the Alternating Projections method (continuous labels, MSE); just the first projections are depicted in the figure: when both sets B and C are convex, and $B \cap C \neq \emptyset$, the algorithm is proved to converge to $z^* = \arg\min_{z \in B \cap C} L(z, y^*)$.

is the function to be minimized. We can recover a projection by taking f as the indicator function over the set C, e.g the indicator function $I_C(y) = 0$ if $y \in C$ and infinite otherwise. In this case, the previous equation becomes

$$z^k = \text{prox}_{\lambda I_C}(y^k) = \arg\min_{\xi} \left\{ ||y^k - \xi||_2^2 \mid \xi \in C \right\}$$

This corresponds to the Euclidean projection onto the feasible set when $\xi \notin C$; conversely, it coincides with the identity operator. For this reason, proximal algorithms can be viewed as generalized projections. The method is guaranteed to converge as long as f is a closed proper convex function. The master step m_β is similar to a proximal algorithm in the sense that we minimize the loss function but restrain to the radius of the ball specified by the parameter β.

- **Data Augmentation**: instead of solving Equation (3.1), we can provide the learner with additional synthetic data representative of the external knowledge and train the model as always. The idea is to bias the ML

model by directly acting on the train set: the advantage is that the learning phase remains unaltered, at the cost of the data manipulation (generating artificial data can be expensive and lead to undesired biases). We can express the method as:

$$h^* = \arg\min_{h \in H} \{L(y^*, h(x)) + L(y', h(x'))\}$$

where we use apices to denote synthetic data and data are generated such that $y' \in C$.

- **Regularization techniques**: regularization techniques are inspired by penalty methods in Constrained Optimization [Ber75]. The regularization term is usually added to the loss function as a penalty term so that it favors the preferred variable configurations, but it has to be crafted carefully to retain a good balance between bias and accuracy of the model. In other terms, by adding a penalty term to the loss function we are moving its minima in the desired direction, which is exactly the trade-off between accuracy and bias. The original problem formulation becomes:

$$h^* = \arg\min_{h \in H} \{L(y^*, h(x)) + L_\rho(y^*, h(x))\}$$

where L_ρ indicates the penalty term.

3.5 Experimental Results

In this Section we describe our experimentation, which is designed around a few main questions:

1. How does the method work on a variety of constraints, tasks, and data sets?
2. What is the effect of the α, β parameters?
3. How does the approach scale?
4. How different is the behavior with different ML models?
5. How does the method compare with alternative approaches?
6. How does the method behave in the long run?

In the following, we will try to give an answer to each question by means of appropriate experiments.

3.5.1 Tasks and Constraints

We experiment on three case studies, both for the classification and regression scenarios. First, we consider a (synthetic) classification problem augmented with a balance constraint, which forces the distribution over the classes to be approximately uniform: this constraint clearly represents a forcing over the original data distribution, but the resulting scenario is well suited to check our algorithm. The classification setting will then be used together with a fairness constraint: this is a realistic constraint that forces the outcome of the predictor to be unbiased with respect to some protected attributes. The third example will involve a fairness constraint in a regression scenario.

Classification

Both experiments will use the Hamming distance (accuracy) (see Table 3.1) as a loss function; the choice to measure accuracy with the Hamming distance, in place of the more popular cross-entropy, is partially due to computational reasons; this metric is indeed a relaxation of the cross-entropy in the case of binary labels. The target space is $\{1..c\}^m$ and the $m_\alpha(y)$ problem is formulated as a Mixed Integer Linear Program (MILP) with binary variables z_{ij}, such that $z_{ij} = 1$ iff the adjusted class for the i-th example is j.

Infeasible master step $m_\alpha(y)$:

$$\min \frac{1}{m}\sum_{i=1}^{m}(1 - z_{i,y_i^*}) + \frac{1}{\alpha m}\sum_{i=1}^{m}(1 - z_{i,y_i}) \tag{3.6}$$

$$\text{s.t.} \sum_{j=1}^{c} z_{ij} = 1 \qquad \forall i = 1..m \tag{3.7}$$

$$\sum_{i=1}^{m} z_{ij} \leq \left\lceil \frac{(1+\xi)m}{c} \right\rceil \qquad \forall j = 1..c \tag{3.8}$$

$$z_{ij} \in \{0, 1\} \qquad \forall i = 1..m, j = 1..c \tag{3.9}$$

The summations in Equation (3.6) encode the Hamming distance w.r.t. the true labels y^* and the predictions y. Equation (3.7) prevents assigning two classes to the same example. Equation (3.8) requires an equal count for each class, with tolerance defined by $\xi > 0$ to ensure the problem admits a solution; we set $\xi = 0.05$ in all our experiments. The balance constraint is stated in exact form, thanks to the discrete labels. The m_α formulation generalizes the

knapsack problem and is hence NP-hard; moreover, since all examples appear in Equation (3.8), no problem size reduction is possible. The m_β problem can be derived from m_α by changing the objective function and by adding the ball constraint as in Equation (3.5).

Feasible master step $m_\beta(y)$:

$$\min \frac{1}{m}\sum_{i=1}^{m}(1 - z_{i,y_i^*}) \tag{3.10}$$

$$\text{s.t.} \sum_{j=1}^{c} z_{ij} = 1 \qquad \forall i = 1..m \tag{3.11}$$

$$\sum_{i=1}^{m}(1 - z_{i,y_i}) \leq \beta \tag{3.12}$$

$$\sum_{i=1}^{m} z_{ij} \leq \left\lceil \frac{(1+\xi)m}{c} \right\rceil \qquad \forall j = 1..c \tag{3.13}$$

$$z_{ij} \in \{0,1\} \qquad \forall i = 1..m, j = 1..c \tag{3.14}$$

Our second use case of study is a classification problem with realistic fairness constraints, based on the Disparate Impact Discrimination Index (DIDI) from [AAV19]:

$$DIDI^c(X,y) = \sum_{k\in K}\sum_{v\in D_k}\sum_{j=1}^{c} d_{kvj} \tag{3.15}$$

$$d_{k,v,j} = \left| \frac{1}{m}\sum_{i=1}^{m} \mathrm{I}[y_i = j] - \frac{1}{|X_{k,v}|}\sum_{i\in X_{k,v}} \mathrm{I}[y_i = j] \right|$$

where K contains the indices of "protected features" (e.g. ethnicity, gender, etc.). D_k is the set of possible values for the k-th feature, and $X_{k,v}$ is the set of examples having value v for the k-th feature. The DIDI indicator measures whether there exists a disparate outcome for examples belonging to protected groups; this gap is null for unbiased models. Note the disparate impact index does not explicitly makes use of the protected features, but measures whether they systematically lead to measurable differences. We can use a Mixed Integer Linear Programming formulation to express both master steps.

Infeasible master step $m_\alpha(y)$:

$$\min \frac{1}{m}\sum_{i=1}^{m}(1 - z_{i,y_i^*}) + \frac{1}{\alpha m}\sum_{i=1}^{m}(1 - z_{i,y_i}) \tag{3.16}$$

$$\text{s.t.} \sum_{j=1}^{c} = 1 \qquad \forall i = 1..m \tag{3.17}$$

$$\sum_{k\in K}\sum_{v\in D_k}\sum_{j=1}^{c} d_{kvj} \leq \epsilon \qquad \forall j = 1..c \tag{3.18}$$

$$d_{kvj} = \left| \sum_{i=1}^{m} \frac{z_{ij}}{m} - \sum_{i\in X_{k,v}} \frac{z_{ij}}{|X_{k,v}|} \right| \tag{3.19}$$

$$z_{ij} \in \{0, 1\} \qquad \forall i = 1..m, j = 1..c \tag{3.20}$$

where Equation (3.18) is the constraint on the DIDI value and Equation (3.19) is then linearized using standard MILP methods. The DIDI scales with the number of examples and has an intrinsic value due to the discrimination in the data. Therefore, we compute $DIDI_{tr}$ for the training set, then in our experiments we have $\epsilon = 0.2 \cdot DIDI_{tr}$. This is again an NP-hard problem defined over all training examples. The m_β formulation can be derived as in the previous case by removing the second term in Equation (3.16) from the loss function and setting it as a constraint.

Regression

Our third case study is a regression problem with fairness constraints, based on a continuous DIDI version from [AAV19]:

$$DIDI^r(X, y) = \sum_{k\in K}\sum_{v\in D_k} d_{kv} \tag{3.21}$$

$$d_{k,v,j} = \left| \frac{1}{m}\sum_{i=1}^{m} y_i - \frac{1}{|X_{k,v}|}\sum_{i\in X_{k,v}} y_i \right| \tag{3.22}$$

As in the classification case, the index measures the dissimilarity between the outcomes of protected features: in the continuous case, this boils down to the comparison between the means of the total distribution and those belonging to different protected realizations. We formulate our regression problem using

the Mean Squared Error (MSE) as a loss function, and the label space is $\mathbb{R}^m$. The m_α problem can be defined via the following Mathematical Program:

Infeasible master step $m_\alpha(y)$:

$$\min \frac{1}{m}\sum_{i=1}^{m}(y_i^* - z_i)^2 + \frac{1}{\alpha m}\sum_{i=1}^{m}(z_i - y_i)^2 \tag{3.23}$$

$$\text{s.t.} \sum_{k\in K}\sum_{v\in D_k} d_{kv} \leq \epsilon \qquad \forall j = 1..c \tag{3.24}$$

$$d_{kv} = \left|\sum_{i=1}^{m}\frac{z_i}{m} - \sum_{i\in X_{k,v}}\frac{z_i}{|X_{k,v}|}\right| \tag{3.25}$$

$$z_i \in \mathbb{R} \qquad \forall i = 1..m \tag{3.26}$$

After a standard reformulation of Equation (3.25), this is a linearly constrained, convex, Quadratic Programming problem that can be solved in polynomial time.

Feasible master step $m_\beta(y)$:

$$\min \frac{1}{m}\sum_{i=1}^{m}(y_i^* - z_i)^2 \tag{3.27}$$

$$\text{s.t.} \sum_{k\in K}\sum_{v\in D_k} d_{kv} \leq \epsilon \qquad \forall j = 1..c \tag{3.28}$$

$$d_{kv} = \left|\sum_{i=1}^{m}\frac{z_i}{m} - \sum_{i\in X_{k,v}}\frac{z_i}{|X_{k,v}|}\right| \tag{3.29}$$

$$\sum_{i=1}^{m}(z_i - y_i)^2 \leq \beta \tag{3.30}$$

$$z_i \in \mathbb{R} \qquad \forall i = 1..m \tag{3.31}$$

The m_β problem, while still convex, is in this case a Quadratically Constrained Problem because of the constraint in Equation (3.30).

3.5.2 Datasets, Preparation, and General Setup

We test our method on seven datasets from the UCI Machine Learning repository [DG17], namely *iris* (150 examples), *redwine* (1,599), *crime* (2,215),

whitewine (4,898), *adult* (32,561), *shuttle* (43,500), and *dota2* (92,650). We use *adult* for the classification/fairness case study, *crime* for regression/fairness, and the remaining datasets for the classification/balance case study.
For each experiment, we perform a 5-fold cross-validation (with a fixed seed). Hence, the training set for each fold will include 80% of the data. All our experiments are run on an Intel Core i7 laptop with 16GB RAM and no GPU acceleration, and we use Cplex 12.8 to solve the master problems. For sake of simplicity, we opted for a straightforward setup of the constraint solver (default parameters, exact solution of even NP-hard problems).
All the datasets for the classification/balance case study are prepared by standardizing all input features (on the training folds) to have zero mean and unit variance. The *iris* and *dota2* datasets are very balanced, while the remaining datasets are quite unbalanced. In the *adult* (also known as "Census Income") dataset the target is "income" and the protected attribute is "race". We remove the features "education" (duplicated) and "native country" and use a one-hot encoding on all categorical features. Features are normalized between 0 and 1. Our *crime* dataset is the "Communities and Crime Unnormalized" table. The target is "violentPerPop" and the protected feature is "race". We remove features that are empty almost everywhere and features trivially related to the target ("murders", "robberies", etc.). Features are normalized between 0 and 1 and we select the top 15 ones according to the `SelectKBest` method of scikit-learn (excluding "race"). The protected feature is then reintroduced.

3.5.3 Parameter tuning

We know that extreme choices for α and β can dramatically alter the method behavior, but not what effect can be expected for more reasonable values. With this aim, we perform an investigation of the impact of α and β by running the algorithm for 15 iterations (used in all experiments), with different parameter values. As a ML model, we use a fully-connected, feed-forward Neural Network (NN) with two hidden layers with 32-Rectifier Linear Units. The last layer has either a SoftMax activation (for classification) or Linear (for regression). The loss function is respectively the categorical cross-entropy or the MSE. The network is trained with 100 epochs of RMSProp in Keras/Tensorflow 2.0 (default parameters, batch size 64).
The results are in Table 3.2. We report a score (row S, higher is better) and a level of constraint violation (row C, lower is better). The S score is the accuracy for classification and the R2 coefficient for regression. For the balance

constraint, the C score is the standard deviation of the class frequencies; in the fairness case studies, we use the ratio between the DIDI of the predictions and that of the training data. Both indicators are then normalized over the constraint satisfaction threshold and capped at 1 for readability (capped values are marked as 1^+). Cells report mean and standard deviation for the 5 runs. All columns labeled with α and β values refer to our method with the specified parameters. The *ideal case* refers to a simple projection of the true target y^* on the feasible space C. This corresponds to an upper bound on the performance of a constrained learner: it exactly matches the constraint threshold while minimizing the loss function, however, it doesn't need to be feasible for the learner, for we don't know the bias set B. The *ptr* column reports the results of the pretraining step, as defined in Algorithm 1, i.e. a constraint-agnostic behavior. Our method lies in between the two extreme cases. Accuracy comparisons are considered fair only for similar constraint violation scores.

The Moving Targets algorithm can *significantly improve the satisfaction of non-trivial constraints*: this is evident for the unbalanced datasets *redwine*, *whitewine*, and *shuttle* and all fairness use cases, for which feasible (or close) results are almost always obtained. As one can expect, satisfying very tight constraints (e.g. in the unbalanced dataset) comes at a steep cost in terms of accuracy. From this point of view, Moving Targets tries to get the most accurate result while satisfying the imposed constraints. Finally, *reasonable parameter choices have only a mild effect on the algorithm behavior*, thus simplifying its configuration. Empirically, $\alpha = 1, \beta = 0.1$ seems to works well and is used for all subsequent experiments.

NN (α, β)		Ptr	$\alpha = 1$ $\beta = .01$	$\alpha = 1$ $\beta = .05$	$\alpha = 1$ $\beta = .1$	$\alpha = .1$ $\beta = .01$	$\alpha = 0^+$ $\beta = 0.1$	Ideal case
Iris	S	$.970 \pm .002$	$.99 \pm .01$	$\mathbf{.997} \pm .004$	$\mathbf{.997} \pm .004$	$.99 \pm .02$	0.995 ± 0.008	$.9968 \pm .0004$
	C	$.23 \pm .08$	$.08 \pm .3$	$.0 \pm .3$	$.0 \pm .3$	$.15 \pm .4$	$.0 \pm .3$	$.0 \pm .3$
Redwine	S	$.709 \pm .005$	$.508 \pm .006$	$\mathbf{.511} \pm .009$	$.506 \pm .006$	$.484 \pm .007$	$.50 \pm .01$	$.525 \pm .002$
	C	$.05 \pm .05$	$.0 \pm .05$	$.0 \pm .03$	$.0 \pm .04$	$.0 \pm .02$	$.0 \pm .05$	$.0 \pm 0$
Whitewine	S	$.644 \pm .002$	$\mathbf{.446} \pm .006$	$.437 \pm .009$	$.439 \pm .009$	$.40 \pm .02$	$.401 \pm .009$	$.524 \pm .002$
	C	$1^+ \pm .2$	$.0 \pm .1$	$.0 \pm .3$	$.0 \pm .2$	$.0 \pm .3$	$.0 \pm .3$	$.0 \pm .1$
Shuttle	S	$.999 \pm 0$	$.39 \pm .04$	$.37 \pm .01$	$.375 \pm .007$	$.37 \pm .03$	$.37 \pm .03$	$.3608 \pm .0008$
	C	$1^+ \pm 0$	$1^+ \pm 1$	$.7 \pm .2$	$.6 \pm .4$	$1^+ \pm 1$	$1^+ \pm 1$	0 ± 0
Dota2	S	$.686 \pm .002$	$.666 \pm .007$	$.661 \pm .002$	$.66 \pm .01$	$\mathbf{.672} \pm .004$	$.656 \pm .006$	$.9984 \pm .0009$
	C	$1^+ \pm .3$	$.6 \pm 1$	$.6 \pm 1$	$1^+ \pm 1$	$.0 \pm .2$	$1^+ \pm 1$	$.0 \pm 0$
Adult	S	$.867 \pm 0.001$	$.818 \pm .005$	$\mathbf{.86} \pm .02$	$.841 \pm .006$	$.852 \pm .004$	$.84 \pm .02$	$0.992 \pm .0005$
	C	$1^+ \pm .2$	$.0 \pm .2$	$.0 \pm .1$	$.1 \pm .4$	$.1 \pm .2$	$.1 \pm .2$	$0. \pm 0$
Crime	S	$.56 \pm .02$	$\mathbf{.49} \pm .01$	$.46 \pm .04$	$.48 \pm .03$	$.45 \pm .05$	$.46 \pm .06$	$.910 \pm .007$
	C	$1^+ \pm .1$	$.1 \pm .4$	$.0 \pm .4$	$.0 \pm .5$	$.0^{\pm}.1$	$.05 \pm .2$	$.0 \pm 0$

Table 3.2: Effect of parameters α and β on different data sets. Each row corresponds to a data set and the performance of the associated model in score (S) and constraint satisfaction (C).

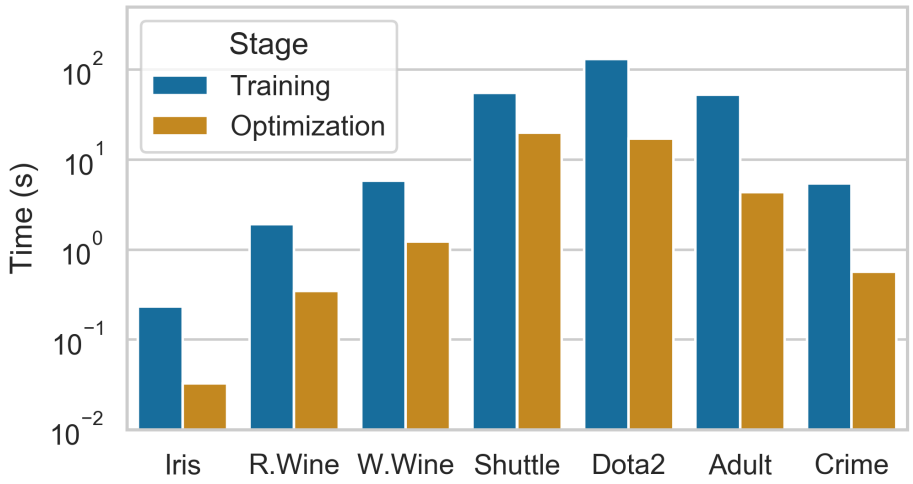

Figure 3.5: Average computational time required by the master step, compared to NN training

3.5.4 Alternative Approaches

Here we describe the setup of alternative approaches that will be used for comparison. Namely, we consider the regularized linear approach from [Ber+17], referred to as RLR, a Neural Network with Semantic Based Regularization [DGS17], referred to as SBR, and the Lagrangian approach from [Cot+19], referred to as TFCO. The first two approaches introduce constraints as regularizers at training time. Their loss function is in the form:

$$L(f(X;\theta), y^*) + \mu g(f(X;\theta)) \tag{3.32}$$

The regularization term must be differentiable and the multiplier μ needs to be hand-tuned. The TFCO approach is similar, but it optimizes both the model parameters and the multipliers by alternating loss minimization and constraint satisfaction.
We use SBR only for the case studies with the balance constraint, which we are forced to approximate to obtain differentiability:

$$g(f(X;\theta)) = \max_{j=1..c} \sum_{i=1}^{m} f(X;\theta) \tag{3.33}$$

		μ	0.01	0.1	1
SBR	Iris	S	**0.984**	0.97	0.4
		C	0	1	1^+
	Redwine	S	0.15	0.15	**0.17**
		C	1^+	1^+	1
	Whitewine	S	0.17	**0.15**	0.14
		C	1^+	0.3	1
	Shuttle	S	0.7	**0.31**	0.14
		C	1^+	0.8	0.8
	Dota2	S	**0.61**	0.48	0.49
		C	1^+	1^+	1^+
RLR	Adult	S	**.83**	.75	.75
		C	1^+	1^+	1^+
	Crime	S	.39	**0.30**	**0.30**
		C	1	0	0

Table 3.3: Effect of parameter μ in regularization methods

i.e. we use the sums of the NN output neurons to approximate the class counts and the maximum as a penalty; this proved superior to other attempts in preliminary tests. The L term is the categorical cross-entropy.

Our SBR approach relies on the NN model from the previous paragraphs. Since access to the network structure is needed to differentiate the regularizer, SBR works best when all the examples linked by relational constraints can be included in the same batch. When this is not viable the regularizer can be treated stochastically (via subsets of examples), at the cost of additional approximation. We use a batch size of 2,048 as a compromise between memory usage and noise. The SBR method is trained for 1,600 epochs.

The RLR approach relies on linear models (Logistic or Linear Regression), which are simple enough to consider a large group of examples simultaneously. We use this approach for the fairness use cases. In the *crime* (regression) dataset L is the MSE and the regularizer is simply Equation (3.22). In the *adult* (classification) dataset L is the cross-entropy; the regularizer is Equa-

tion (3.15), with the following substitution:

$$d_{k,v,j} = \left| \frac{1}{m} \sum_{i=1}^{m} \theta^\top x_i - \frac{1}{|X_{k,v}|} \sum_{i \in X_{k,v}} \theta^\top x_i \right|$$

This is an approximation obtained according to [Ber+17] by disregarding the sigmoid in the Logistic Regressor to preserve convexity. We train this approach to convergence using the CVXPY 1.1 library (with default configuration). In RLR and SBR classification, the introduced approximations *permit to satisfy the constraints by having an equal output for all classes*, i.e. completely random predictions. This undesirable behavior is countered by the L term.
The results of a hand-tuning process for SBR and RLR are reported in Table 3.3. In most cases, larger μ values tend as expected to result in better constraint satisfaction, with a few notable exceptions for classification tasks (*iris*, *dota*, and *adult*). The issue is *likely due to the approximations introduced in the regularizers*, since it arises even on small datasets that fit in a single mini-batch (*iris*). Further analysis will be needed to confirm this intuition. The accuracy decreases for a larger μ, as expected, but at a rather rapid pace. In the subsequent experiments, *we will use for each dataset the RLR and SBR that offer the best accuracy while being as close to feasible as possible*: these are the cells in bold font in Table 3.3. For the TFCO approach, we use again the NN from previous paragraphs, a minibatch of size 200 and 100 iterations with 200 iterations per loop. The optimizer is ADAM with default parameters. The method is in principle able to reach an optimal solution, but *only in expectation*, at the price of having a stochastic classifier. To enable a fair comparison, we obtain a single classifier using the "best" method from the reference implementation.

		Regularized methods	TFCO	NN	LR	Ensemble trees	NN_{pp}
Iris	S	$.984 \pm .006$	$.95 \pm .003$	$\mathbf{.997} \pm .004$	$.96 \pm .02$	$.995 \pm .004$	$.96 \pm .01$
	C	$.0 \pm 0.2$	$1^+ \pm 1$	$.0 \pm 0.3$	$.1 \pm .4$	$.0 \pm .2$	$.07 \pm .4$
Redwine	S	$.17 \pm .05$	$.3 \pm .2$	$\mathbf{.506} \pm .006$	$.32 \pm .01$	$.40 \pm .02$	$.480 \pm .001$
	C	$.1^+ \pm .5$	$1^+ \pm 1$	$.0 \pm .05$	$.6 \pm .2$	$1^+ \pm .5$	$1^+ \pm .3$
Whitewine	S	$.15 \pm .03$	$.3 \pm .1$	$\mathbf{.439} \pm .009$	$.025 \pm .009$	$.37 \pm .04$	$.47 \pm .02$
	C	$.3 \pm .3$	$1^+ \pm 0$	$.0 \pm .2$	$.8 \pm .2$	$1^+ \pm 1$	$1^+ \pm 1$
Shuttle	S	$.31 \pm .04$	$.2 \pm .3$	$.375 \pm .007$	$\mathbf{.332} \pm .007$	$.51 \pm .05$	$.5 \pm .1$
	C	1 ± 1	$1^+ \pm 0$	$.6 \pm .3$	$.4 \pm .4$	$1^+ \pm .6$	$1^+ \pm 1$
Dota2	S	$.61 \pm .02$	$.53 \pm .01$	$.66 \pm .01$	$.592 \pm .005$	$.53 \pm .01$	$\mathbf{.689} \pm .003$
	C	$1^+ \pm 1$	$1^+ \pm 0$	$1^+ \pm 1$	$.5 \pm 0$	$1^+ \pm 1$	$.0 \pm .8$
Adult	S	$.834 \pm .001$	$.87 \pm .01$	$.841 \pm .006$	$.805 \pm .006$	$.76 \pm .01$	$\mathbf{.865} \pm .003$
	C	$1^+ \pm .2$	$1^+ \pm .05$	$.1 \pm .4$	$.0 \pm .2$	$.0 \pm .2$	$.0 \pm .4$
Crime	S	$.30 \pm .01$	$\mathbf{.58} \pm .05$	$.48 \pm .03$	$.369 \pm .008$	$.49 \pm .01$	$.484 \pm .008$
	C	0 ± 0	$.0 \pm .1$	$.0 \pm .5$	$.0 \pm 0$	$.2 \pm .05$	$.0 \pm .1$

Table 3.4: Benchmarks between Moving Targets with different ML models and alternative approaches on several data sets

We can now compare the performance of Moving Targets using different ML models with that of the alternative approaches presented above, plus a preprocessing approach adapted from [KC09], referred to as NN_{pp} and obtained by setting $\alpha, \beta \to \infty$ in our method.
For our method, we consider the following ML models: 1) the NN from the previous section with $\alpha = 1, \beta = 0.1$; 2a) a Random Forest Classifier with 50 estimators and a maximum depth of 5 (used for all classification case studies); 2b) a Gradient Boosted Trees model, with 50 estimators, maximum depth 4, and a minimum threshold of samples per leaf of 5 (for the regression case study); 4a) a Logistic Regression model (for classification); 4b) a Linear Regression model (for regression). All models except the NN are implemented using scikit-learn [Ped+11]. In Table 3.4, the tree ensemble methods are reported on a single column, while another column (LR) groups Logistic and Linear regression.
Our algorithm seems to work well with all the considered ML models: tree ensembles and the NN have generally better constraint satisfaction (and higher accuracy for constraint satisfaction) than linear models, thanks to their larger variance. The preprocessing approach is effective when constraints are easy to satisfy (*iris* and *dota2*) and on all the fairness case studies, though less so on the remaining datasets. All Moving Targets approaches tend to perform better and more reliably than RLR and SBR. The case of RLR and LR is particular, since in principle the two approaches can be expected to behave identically (convex problem and same constraint formulation): the gap is due to an incomplete exploration of the space of the multiplier μ. The example emphasizes a practical problem that often arises when dealing with regularized loss functions: the value of the multiplier has to be thoroughly calibrated by hand, while Moving Targets allows to directly define the desired constraint threshold and is quite robust to different parameter values.

3.5.5 Scalability and Convergence

We next turn to investigate the method scalability. Our examples can be considered worst cases, since all examples appear in the single constraints and in some cases involve NP-hard problems. We report the average time for a master step in Figure 3.5, with the average time for a learner step (100 epochs of our NN) for reference. At least in our experimentation, *the time for a master step is always very reasonable*, even for the *dota2* data set for which we solve NP-hard problems on 74,120 examples. This is mostly due to the clean

structure of the m_α and m_β problems. Of course, for sufficiently large training sets, exact solutions will become impractical and non-exact optimization will need to be considered (e.g. meta-heuristics or matheuristics). However, the structure of the algorithm allows for heuristic resolution approaches, which can address the computational burden of very large data sets.

The convergence properties of Moving Targets are verified experimentally, by running the method for up to 40 iterations; results are shown in Figure 3.6. We run the experiment for two tasks: classification on the redwine data set with balance constraint and regression on the crime data set with fairness constraint. The comparison shows both score and constraint satisfaction performances for three different ML models: LR, RF (GB in the regressive scenario), and NN. As we can see, the NN model is the one with the highest variance, although it achieves the best performances in the classification task, stabilizing after roughly 20 iterations. The LR model, on the other hand, requires a very small amount of iterations to converge, as well as the ensemble tree methods. Notice that in the regressive task all ML methods reach convergence in just a few iterations: this is due to the convexity of the problem.

3.5.6 Generalization

Since our main contribution is an optimization algorithm, we have focused so far on evaluating its performance on the training data, as it simplifies its analysis. We now assess its performance on the test data. In addition to the models of the previous paragraphs, we consider a Random Forest with very low bias (100 estimators with no depth limit), denoted as LBRF, simply trained over the *ideal case* results. Due to the low bias, even this simpler training method obtains feasibility and matches closely the accuracy of the ideal case on the training set.

The results of this evaluation are reported in Table 3.5, in the form of the average ratio between the scores and the level of constraint satisfaction in the test and the train data. With a few exceptions (e.g. satisfiability in *iris*), the models generalize well in terms of both accuracy and constraint satisfaction. Given the tightness of some original constraints and the degree to which the target was altered, this is a remarkable result. The simpler LBRF approach performs poorly on the test set: while the low bias simplifies training, the price to pay in terms of lack of generalization is quite steep.

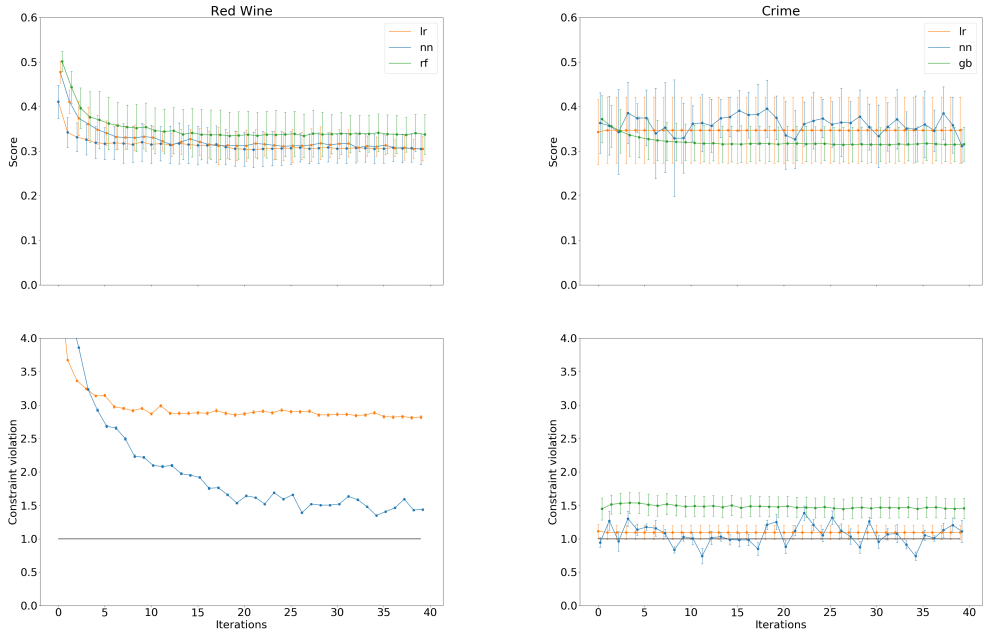

Figure 3.6: Long-run results for different ML models and tasks: on the left, balance constraint on the Red Wine data set; on the right, fairness constraint on the Crime data set. We report score and constraint satisfaction (top and bottom) over 40 iterations.

3.6 Conclusion and Future Work

In this Chapter we have presented and tested Moving Targets, a decomposition approach designed to augment a generic supervised learning algorithm with constraints, by iteratively adjusting its training labels. The method is devised to prioritize constraint satisfaction over accuracy and proved to behave well on a selection of tasks, constraints, and datasets. Its relative simplicity, reasonable scalability, and the ability to handle any classical ML model and any state-of-the-art constraint solver make it well suited for use in real-world settings.

Many open questions remain: we highlighted limitations of regularization-

		NN	Ens. Trees	LR	LBRF
Iris	S_{ts}/S_{tr}	0.96	0.96	0.99	0.96
	C_{ts}/C_{tr}	5.68	5.17	4.31	5.16
Redwine	S_{ts}/S_{tr}	0.62	0.92	0.94	0.72
	C_{ts}/C_{tr}	1.22	1.04	1.35	2.68
Whitewine	S_{ts}/S_{tr}	0.70	0.96	1.00	0.71
	C_{ts}/C_{tr}	1.11	1.00	0.99	2.92
Shuttle	S_{ts}/S_{tr}	0.99	1.00	0.99	1.02
	C_{ts}/C_{tr}	0.97	1.00	1.01	1.35
Dota2	S_{ts}/S_{tr}	0.83	1.00	0.99	0.58
	C_{ts}/C_{tr}	1.10	1.00	1.03	2.79
Adult	S_{ts}/S_{tr}	0.99	1.00	1.00	0.86
	C_{ts}/C_{tr}	1.55	1.92	0.98	4.21
Crime	S_{ts}/S_{tr}	0.75	0.73	0.93	0.50
	C_{ts}/C_{tr}	0.74	1.05	1.03	1.53

Table 3.5: Generalization of various models in the test scenario

based techniques that deserve much deeper analysis. From a high-level perspective, Moving Targets seems to be better suited to deal with relational constraints, or constraints defined over (part of) the input examples, while regularization techniques constitute a more viable option when it comes to shape constraints (e.g. monotonicity of the predictor with respect to one or more input feature). The convergence properties of our method still need to be properly characterized. The method scalability should be tested on larger datasets (for which using approximate master steps will become necessary), so as to assess the effect of using meta-heuristics or matheuristics. Moreover, since we allow the use of any ML model, it may be interesting to *combine* Moving Targets with other approaches for constraint injection in ML.
As for future perspectives, Moving Targets has already been extended to a broader spectrum of constraints, including global inequalities and fairness measures such as Equal Opportunity and Equalized Odds. In a joint project with a partner University, the method was implemented in the AI Domain Definition Language (AIDDL), which provides a modeling language and framework for integrative AI[1]. This results in a modular framework that makes it possible to use Moving Targets in a wider system, able to provide custom AI

[1]see https://www.ai4europe.eu/research/research-bundles/ai-integration-languages

models. By specifying the type of task (i.e. classification or regression), the learning model, and additional constraints, the end-user can shape a custom and human-oriented predictive model.

Chapter 4

Applications

This Chapter presents two practical applications that were done together with Optit within larger business projects. They both involve, at different levels, an example of integration of a data-driven model and a Combinatorial Optimization problem: in the first case, a Survival Analysis method is used to estimate the fault probability associated with water pipelines, starting from historical data. The model predictions are then incorporated in the formulation of the optimization model via offline generation of a failure probability table. Although not involving a Machine Learning model, we can regard this application as a general example of leveraging data-driven models for the modeling of a COP, as seen in the paragraph on learner embedding in Section 2.2.1. Results of this work are published in [Poz+21].

In the second experiment, the problem at hand is a three-dimensional Bin Packing Problem (3D-BPP). We use a learning model to estimate the number of bins required to pack a set of items, given a problem instance. Because of the scarcity of realistic instances, we resort to synthetic records to enlarge our data set; the learner is then trained in a supervised manner over the resulting data set, where each synthetic instance is paired with the corresponding solution of the 3D-BPP problem. Finally, the bin estimator is integrated within the resolution algorithm to infer promising (partial) solutions and hence guide the process. Depending on how the integration is carried out, the approach can be classified as either a *search guidance* method or a *bounding mechanism*, as

seen in Section 2.2.2.

4.1 Predictive Maintenance

Maintenance is a key activity in industrial environments, for it involves costs that can heavily affect the business and malfunctioning that can lead to major issues; indeed, the failure of a single component might be associated not only with the cost required for its replacement but may also cause system downtime, potentially resulting in a serious penalization of the operations and customer disservices.

There are three leading maintenance strategies: *Reactive Maintenance*, *Preventive Maintenance*, and *Predictive Maintenance*. *Reactive Maintenance*, as the name suggests, is concerned with maintenance operations planned as a consequence of failures or when the malfunctioning level reaches a certain threshold. This is a *run-to-failure* strategy that exploits the system elements until they reach the maximum production output. However, the cost of repairing or replacing a component, together with its side effects on other parts of the systems or downtime penalization, could overcome the savings resulting from running it to failure.
A safer approach is constituted by *Preventive Maintenance*, where components are replaced on a regular basis by assigning to each component an expected useful lifetime, which is estimated according to the historical malfunctions. In practice, this results in a *better-safe-than-sorry* strategy, with potential unnecessary maintenance operations; in addition, unexpected failures due to faulty components may still occur.
Predictive Maintenance, on the other hand, aims at exceeding the shortcomings of the above strategies by estimating the best time for replacement interventions. The idea is to collect real-time data describing the component behavior and use it to identify critical situations. This latter strategy combines the advantages of reactive and preventive maintenance: by monitoring the state of each component, its replacement is scheduled such that its remaining useful life is minimal and failures are avoided. This requires both accurate sensors to track the state of the system and sound models to predict a malfunction before it occurs. It goes without saying that Predictive Maintenance can be affected by unexpected events: by definition, these cannot be completely avoided, however, the policy tries to maximize the production output of each component.

4.1.1 Problem Overview

A major Italian multi-utility operating in the city of Brescia detains a system for District Heating. The latter is constituted by a network of pipes, laid in the 70s, and related infrastructures and provides the customers with hot water. Good management of such a network is of paramount importance, both for the company's revenue and for the user service. In fact, failures due to pipe breaking implicate water leaking, customer dissatisfaction, and the costs associated with either repairing or replacing the network component.
The network features over 650 km long pipelines, serving more than 20,000 consumer points and covering almost 60% of the heat demand of the city. A wide range of piping technologies is present, from newer pre-insulated pipes (PR) to older traditional hooded pipes, characterized by different installation types (e.g. non-pre insulated steel, fiber-cement sheath, wanit, etc). Another factor to take into account is the presence of operational manholes, which are network elements functional to interventions on pipes with old technologies that require expensive upkeep operations.

Thanks to forward-thinking practices, the company has recorded every intervention on the pipeline since the late 70s, allowing for a historical analysis of failures. In fact, while the accurate knowledge of the composition of the network is obviously valuable, it cannot provide, alone, definitive information about potential future failures, which is key for the long-term planning of maintenance operations. To this end, we will exploit the available records of failure events from the year 1999 to the year 2016. Each failure event is characterized by the year of occurrence, the year of the initial installation of the affected pipe, its geographical coordinates, technology, and other physical properties. This data constitutes only a partial representation of the full evolution of the network; since pipes could be either fixed or replaced and most records specify the occurrence of the event, but not necessarily how it was addressed. Yet, for simplicity's sake, it can be assumed that each failure originated a replacement that fully restored functionalities as if the asset was fully renewed.

4.1.2 The approach

Given the historical data set of recorded failures and the cartographic representation of the pipeline, we address the problem of maintenance planning in three steps:

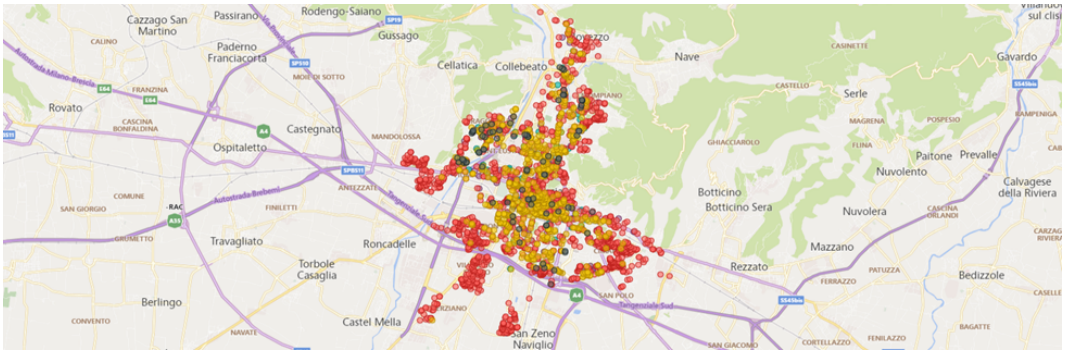

Figure 4.1: Geographic Distribution of the recorded failure events. Colors correspond to the technology of the pipe.

1. *Network Aggregation*: the input structure of the network comprises a large number of pipes. This has two drawbacks: on one hand, when the network is fragmented and composed of many pipes, it is more likely that interventions will involve small pipes, which may not justify the opening of a construction site (since this causes several disservices); on the other hand, we will later see that the optimization model scales with the number of pipes, thus becoming of impractical size when this number is large. For these reasons, the input network will be aggregated to create a simpler and more homogeneous representation.

2. *Risk Analysis*: by comparing the aggregated network with the historical data of failures, a statistical analysis is used to estimate the risk of failure associated with each pipe of the network.

3. *Optimization*: once the network has been aggregated and an estimator for the pipe failure risk is available, we have all the ingredients to formulate an optimization problem. The objective is to minimize the failures on the pipeline over a given planning horizon.

Network Aggregation From a cartographic data point of view, the network provided is too fine-grained and heterogeneous, both for analysis and maintenance planning purposes. Thus, the network has been homogenized by coalescing adjacent pipes with compatible characteristics and subsuming customers' feeding branches by their respective backbone pipes, bringing the original network from 30,000 pipes to 5,000 pipes. This allows a much more manageable data set. In particular, adjacent pipes are aggregated if they share

the same values of diameter, installation year, and type of installation. The resulting graph implicitly avoids maintenance interventions involving short pipes that are to be penalized because unfavorable.

Risk Analysis The goal of Risk Analysis is to provide an estimator able to predict the probability of failure associated with each pipe in the network. Indeed, in order to devise a meaningful plan for maintenance operations, we need a measure for the effect of the failures or, from a complementary perspective, the effect of the maintenance.
To this end, we analyze the data of historical failures and track the impact of pipe features: in particular, we consider the diameter of the pipe, the installation year, the technology, and the geographical risk. The geographical risk is the result of an ad-hoc analysis that assigns a score to different areas of the city of Brescia, attempting to identify regions where pipes are more likely to undergo failure events.
Moreover, the data containing the history of failures presents some characteristics that might hinder the analysis process: as is usual for historical data, we have records of failures only for a limited period of time, in particular from the year 1999 to the year 2016. This leads to the following scenarios:

- there are pipes for which we know both the installation year and the failure year;
- there are pipes for which we know neither the installation year nor the failure year because both values fall outside of the observed interval (these are referred to as *left-truncated*, see [KM03])
- there are pipes for which the installation year is known, but hasn't experienced any failure yet (these are referred to as *right-censored*, see [KM03])

A variety of approaches can be applied to devise a risk estimator based on the input data: the two main categories are represented by ML models and Survival Analysis (also named Stochastic Survival Models [KM03]). We opt for the latter, which is specifically made to deal with the kind of data at our disposal, results easier to interpret and analyse, and has already been employed in cases similar to our [Chr11].

We briefly overview the basic equations of Survival Analysis: given the probability density function f of failure events, the associated cumulative distribution function F is given by

$$F(t) = \int_0^t dx\, f(x) \tag{4.1}$$

From this equation, it is possible to define the *survival function* S, which represents the probability that the failure event has not occurred by time t and is therefore obtained as the complement of F

$$S(t) = \int_t^\infty dx\, f(x) = 1 - F(t) \tag{4.2}$$

Our analysis boils down to estimating the value of the function S, which in turn allows to model the optimization problem. From a practical perspective, several techniques exist to estimate the value of Equation (4.2); for the sake of simplicity we choose the non-parametric Kaplan-Meier estimator [KM03], which can be defined as

$$\hat{S}(t) = \prod_{i=0}^{t} \left(1 - \frac{d_i}{r_i}\right) \tag{4.3}$$

where d_i is the number of failures that occurred at time t_i and r_i is the number of pipes at risk at the same time instance, i.e. pipes still functioning just before t_i.
It is important to point out that Equation (4.3) considers only the time component, regardless of the remaining features characterizing the pipes. In order to handle such covariates, it is necessary to estimate separate functions (for instance, one for each technology type).

To evaluate the impact of each variable in assessing the risk, we performed various experiments and estimate survivor functions with different groupings of the pipes, with varying feature values. We assessed the resulting groupings through the log rank test [BA04], used to test the null hypothesis against potential differences between the populations of pipes in the probability of a failure event. Furthermore, we verified the resulting risks estimate on a test sample of pipes with the aid of domain experts. Interestingly, in many cases the log rank test pointed out a significant difference between the groups (e.g., different technologies showed different behavior); however, an in-depth

analysis showed that the differences could have been the result of biased and anomalous data. A confrontation with domain experts confirmed the latter interpretation.
The result of the evaluation process led us to choose the hazard curves shown in Equation (4.2), which consider only the geographical risk as a discriminant factor for the estimation of failure probability. These are simple models with a straightforward interpretation and, as confirmed by experts, give reasonable insights to assess the probability of pipe malfunctioning.

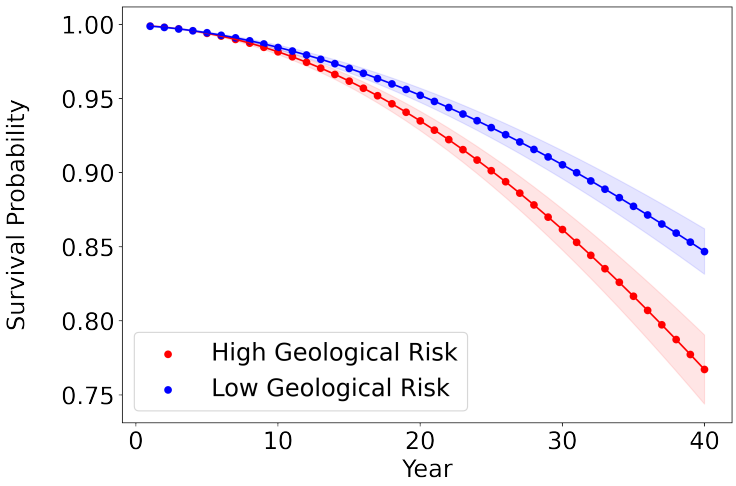

Figure 4.2: Estimated survival probability function used for predicting maintenance planning: the two curves correspond to different value of the pipe geographical risk.

Optimization In this paragraph, we give a formal description of the optimization model used. We want to devise the optimal plan of the intervention operations on the network, such that the costs involved are minimized (e.g. maintenance cost, intervention cost, cost due to disservices, etc). We choose a time granularity corresponding to years, in accordance with the historical data provided. This sets up the discretization of the model decision over time: in particular, we plan the operations over a time horizon of Y_{plan} years such that the Net Present Value (NPV) of the related cost is minimized over the next Y_{cost} years. That is, the first interval defines when the interventions can take place, while the second is used to compute the final revenues. Two types of elements constitute the network: pipes and manholes. A pipe $p \in P$ is

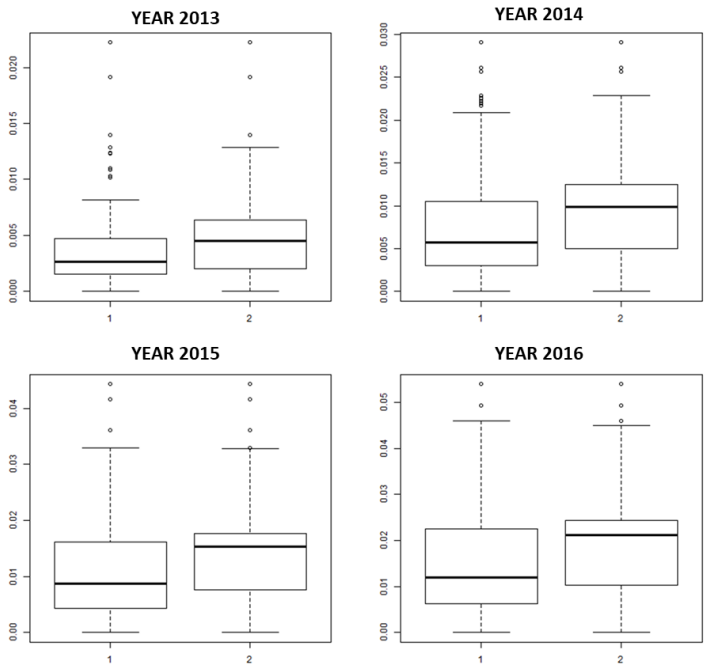

Figure 4.3: Survival analysis validation for the years 2013 to 2016. Labels 1 and 2 denote the distribution of the risk-index for non-faulty and faulty pipes, respectively.

characterized by its diameter d, length l, technology t and age a. A manhole $m \in M$ is uniquely characterized by a risk factor r, which is directly related to the number of years after which the manhole will require maintenance works; these elements were conceived to facilitate interventions on the pipes, but are no longer required thank to modern technologies and should be removed from the network whenever possible as they only represent a cost. To state the decision support model, we need to declare all the expenses involved in the process; the company provided us with several tables of the relevant costs, that we briefly describe and characterize in the following.

A pipe $p \in P$ is associated with

- a *maintenance cost* $c_m(p)$, corresponding to the cost for repairing the pipe after a failure event;
- a *substitution cost* $c_s(p)$, required for its replacement with a new one;

- a *malfunctioning cost* $c_f(p)$, corresponding to the cost of leaking and other disservices following a failure event;
- a *reduced heat loss* $c_h(p)$, due to the replacement of the pipe with a new one with better thermal insulation.

Likewise, a manhole $m \in M$ is associated with

- a *demolition cost* $c_d(m)$;
- a *maintenance cost* $c_n(m)$, directly related to its risk factor.

The optimization model has to decide if and when to replace a pipe or demolish a manhole over the time horizon Y_{plan}, taking into account the trade-off between the benefits connected to a newer pipe and its substitution cost: to do so, we compute for each year the corresponding total network cost, that represents the sum of all the expected costs over pipes and manholes. We represent the average charge for a pipe p and a cost term c, by multiplying its failure probability on the year i, $\lambda^{(i)}(p)$, by the specified expense. The same reasoning can be done to compute the costs associated with the set of manholes. It is clear that the costs computed in this way become a good statistical approximation of the true value when the network size is big enough; a direct verification with domain experts ensured our computed expected costs resulted close to their annual expenses.

The probability $\lambda^{(i)}(p)$ can be obtained from the survival function of Equation (4.2) by imposing the condition that the pipe breaks exactly in the time interval comprised between year i and the following, given its age a. Denoting by T the time at which the failure event takes place, we have

$$\begin{aligned} \lambda^{(i)}(p) &= F(T \geq a+i, T < a+i+1 \mid T > a) \\ &= \frac{S(a+i) - S(a+i+1)}{S(a)} \end{aligned} \tag{4.4}$$

The function $\lambda^{(i)}(p)$ is represented in Figure 4.4: notice that the effect of the pipe age a is that of shifting and rescaling the probability function, as can be seen also in Equation (4.4).
In a similar way, the costs associated with a manhole are distributed linearly over the years, knowing that the risk factor is proportional to the number of years after which maintenance works will become necessary.

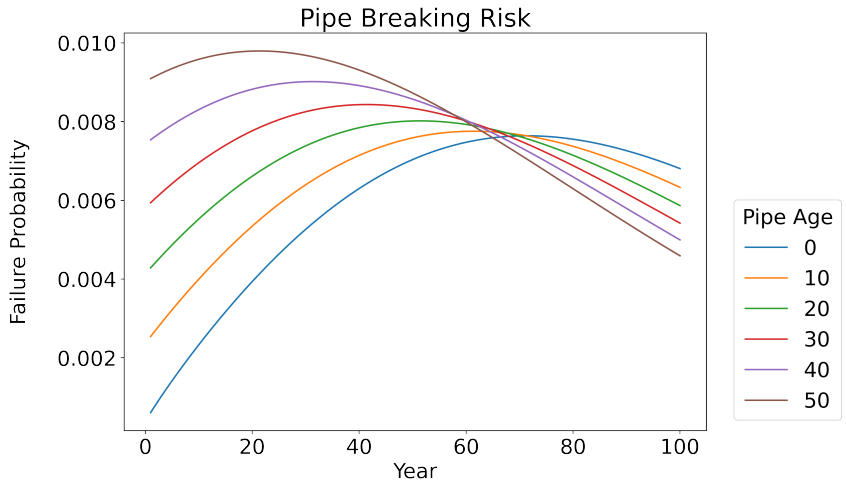

Figure 4.4: Estimated failure probability function for maintenance planning: the different curves correspond to pipes of different ages.

An element replacement is modelled with a Boolean decision variable: a pipe $p \in P$ is represented by the variable $x_p^{(i)} \in \{0, 1\}$, $i \in \{1, \ldots, Y_{plan}\}$, which has non-zero value if the pipe p is replaced during year i. Likewise, a manhole $m \in M$ is represented by a variable $y_m^{(i)} \in \{0, 1\}$, $i \in \{0, \ldots, Y_{plan}\}$.

We now have all the elements to state the costs sustained over the interval Y_{cost}. For each cost invoice, we distinguish two separate addends: the expenses due to non-replaced elements and those due to replaced ones. For the set of pipes and a generic cost term c, we can write

Costs associated with non-replaced pipes

$$\xi_{nr}^{(i)}(c) = \sum_{p \in P} c(p) \cdot \lambda^{(i)}(p) \left(1 - \sum_{k=1}^{\min(i, Y_{plan})} x_p^{(i)}\right) \tag{4.5}$$

Costs associated with replaced pipes

$$\xi_{r}^{(i)}(c) = \sum_{p \in P} \sum_{k=1}^{\min(i, Y_{plan})} c(p) \cdot \lambda^{(i-k)}(p) x_p^{(i)} \tag{4.6}$$

By substituting the generic cost term c with the proper specific cost, described in Section 4.1.2, we obtain all the needed terms: the total maintenance cost for each year for both replaced and non-replaced pipes, $C^{(i)}_{m,r} = \xi^{(i)}_r(c_m)$ and $C^{(i)}_{m,nr} = \xi^{(i)}_{nr}(c_m)$, the malfunctioning costs of pipes $C^{(i)}_{f,r} = \xi^{(i)}_r(c_f)$ and $C^{(i)}_{f,nr} = \xi^{(i)}_{nr}(c_f)$ and their reduced heat loss $C^{(i)}_{h,r} = \xi^{(i)}_r(c_h)$. In analogous way, by substituting $x^{(i)}_p$ with $y^{(i)}_m$ in Equation (4.5), we obtain the maintenance cost for the non-demolished manholes $C^{(i)}_{n,r}$. The total investment cost in year i amounts to the sum of the cost due to the substitution of the pipes and the demolition of the manholes

$$C^{(i)}_s = \sum_{p \in P} c_s(p)c \cdot x^{(i)}_p + \sum_{m \in M} c_d(m) \cdot y^{(i)}_m \tag{4.7}$$

The objective function f to be minimized is expressed as the sum of all costs undertaken over the revenue interval Y_{cost}, discounted by the company's cost of capital

$$f(x, y) = \sum_{i=1}^{Y_{cost}} \frac{z^{(i)}(x, y)}{(1 + w)^i} \tag{4.8}$$

$$z^{(i)}(x, y) = C^{(i)}_s + C^{(i)}_{m,r} + C^{(i)}_{m,nr} + C^{(i)}_{n,r} + C^{(i)}_{f,r} + C^{(i)}_{f,nr} + C^{(i)}_{h,r} \tag{4.9}$$

where we fix $C^{(i)}_s = 0 \ \forall i > Y_{plan}$ and w is the company's cost capital. Apart from integrality constraints on the variables $x^{(i)}_p$ and $y^{(i)}_m$, there are additional rules reflecting the company's policies as well as the external regulations. We list some of them below:

1. a pipe is repaired at most once. This goes hand-in-hand with the initial assumption that a pipe can break at most once, which is reasonable for the considered time interval;

2. a manhole can be demolished when all the incident pipes belong to the technology PR. After demolishing, the manhole is removed from the network;

3. there is a specific budget for retrofitting investments and a separate one for maintenance

4. in order to avoid spot interventions, the substitution of pipes has to be planned such that the involved elements exceed a minimum total length. This compensates for the absence of the cost required to open a construction site in the optimization model.

The model obtained is an Integer Linear Programming model (ILP); the number of variables scales linearly with the number of pipes: as stated before, this takes advantage of the network aggregation previously described.

4.1.3 Results

We applied the methodology described in Section 4.1.2 to our use case. First, the risk analysis estimator was validated, simulating to be in 2012 and computing the expected failures for the following years. Figure 4.3 shows boxplots for the results over the years 2013 to 2016: the thicker line is the median value, while the box represents the interquartile range (IQR) between the first and third quartiles. Results indicate how the estimated risk values are higher for pipes that actually experienced a failure event. In fact, the left column of each plot (labeled as '1') corresponds to pipes that did not fail in the specified year, while the right column (labeled as '2') to those that actually failed. As expected, the confidence interval widens as we move further from the reference year; nevertheless, the number of expected failures per year is in line with the trends observed in the previous years.

Once validated the risk estimator, we implemented the optimization model and analyzed the optimal maintenance plans over the revenue horizon. The study had two goals: 1) identify the most critical segments of the network, both in terms of risk index and potential economic impact of an eventual failure and 2) analyze how different drivers influenced the number of expected faults in the network, e.g., what would be the necessary budget to keep the number of faults steady through the years and whether that would be worth it from an economic standpoint. Given a 5-year maintenance plan (i.e. $Y_{plan} = 5$) and a time horizon for revenues of 30 years (i.e. $Y_{cost} = 30$), we benchmarked scenarios with different budget allocations, expressed as percentages with respect to the actual company's resource allocation in the reference year. The *Baseline* scenario represents the case in which no pipe replacement is foreseen and only emergency operations are taken; this corresponds to a *Reactive Maintenance* policy. The objective function of our model considers not only the investment

cost of the piping replacement but also the contribution of the potential inefficiencies caused by failures, which determine water and heat losses as well as operational disservices that may or may not be overcome by regulation strategies. Intuitively, we desire to have a small number of pipe failures and to address high-risk pipes, favoring the substitution of bigger pipes for their malfunctioning has a bigger impact, both in terms of incurred costs and customer disservices. The benchmark in Figure 4.6 highlights how the break-even point with respect to the baseline is shifted forward as the budget increases, reflecting higher initial capital costs. Yet, a higher budget (which triggers a higher replacement ratio) ensured a reduction of the expected fault occurrences, with significant impacts not only from an economic standpoint but also from a quality of service (and customer satisfaction) one. The smaller NPV of the maintenance costs evaluated over 30 years is linked to the BDG-100% scenario and the Baseline featured the highest figure, while the BDG-200% scenario performed halfway, yet with a 2.6% improvement with respect to the Baseline. The payback time was 15 years for BDG-100% and 25 years for BDG-200%, as shown in Figure 4.5. As shown in Figure 4.6, the Baseline scenario (i.e., no replacement) outlined an increase of breakdowns by a factor of 20% in the time interval considered, which could be mitigated by 10% with the current budget (BDG 100%) and be neutralized (i.e., steadying the number of failures) with a threefold increase in the set budget, while even higher figures (1500% of the current budget) would have been necessary to halve the number of failures in a 5-year span. Nevertheless, these scenarios proved to be not economically convenient, because the 30-year horizon is not enough to recover such high investments. Moreover, a large budget also meant intervening in less significant areas, where the risk factors are lower, thus less impactful. However, it is to be pointed out that some cost sources are excluded from the current computation, for instance, incentives and discounts that may be granted to companies to modernize their infrastructures.

4.2 Bound Estimation

In this second part of the Chapter we present an experimentation on a practical case of COP; our experimentation has two goals: 1) learn a fast approximation of the objective function of an optimization problem and 2) use the learned model for bounding within the resolution process to boost the performances. The approach is very similar to [FF19], where a ML model is used to estimate

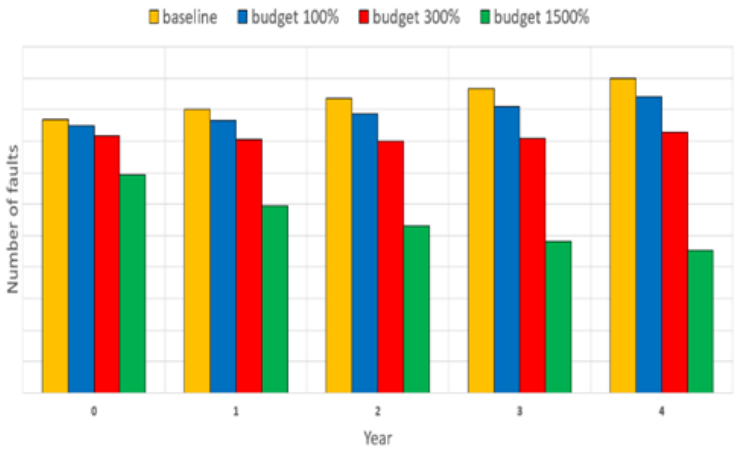

Figure 4.5: Expected number of pipe faults over the years, for different budget allocations, as a result of maintenance planning.

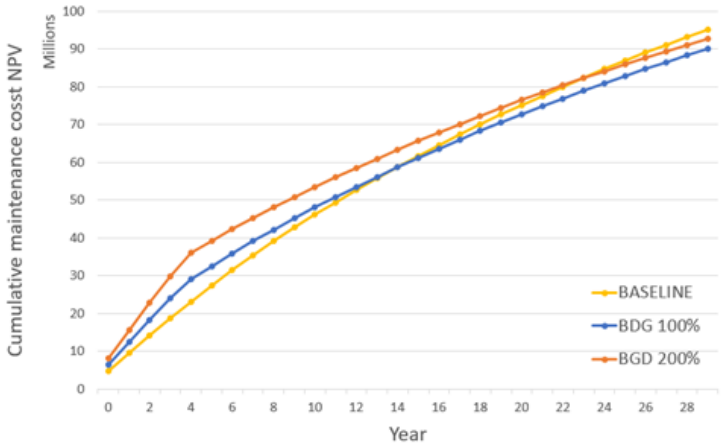

Figure 4.6: Cumulative maintenance costs for different budget allocations, expressed as percentages of the current baseline, as a result of maintenance planning.

the objective function of an optimization problem without solving it, but our data generation is more general and we then integrate the learning model in the resolution process. This second part is similar to [HTT20], where a tree search exploration is combined with a DNN to learn novel heuristic strategies; however, we build on a heuristic resolution method based on creation and destruction operators. Other works tackle the 3D-BPP through Reinforcement Learning [Zha+20], or by devising data-driven tree search strategies [Zhu+21].

4.2.1 Problem Overview

A major company operating in the logistic sector delivers goods by means of several vectors, including airplanes. The cost related to flights makes the number of airplanes one of the major cost drivers, therefore it is crucial to efficiently manage the number of goods that are transported by each flight, in order to minimize the airplanes used. On a more atomic scale, items are first loaded on pallets, i.e. bins of fixed size, and then the pallets are loaded into airplanes: the goal is then to load a given item set on the least possible number of pallets.

The resulting optimization model is a three dimensional Bin Packing Problem (3D-BPP) [MPV00], a well-known problem in the literature of COP. As is often the case with real-world processes, this 3D-BPP has many custom constraints that reflect either company policies, structural constraints, or item-specific constraints.

However, here we are not concerned with solving the optimization problem per se, for which a heuristic-based solver is available, but rather in studying whether a learner is able to infer some major characteristics of the problem solution, in particular the number of bins required to load a given set of goods. In fact, such learner can be used for two reasons: 1) having a quick estimation of the bins required to load the products facilitates the loading organizational process, and 2) it can be used to provide an estimated bound on the number of bins of the problem solution and thus boost the performance of the resolution process.

We will first give an overview of the methodological approach employed, then evaluate the performance of ML models in estimating the number of bins, and finally, propose a couple of algorithmic customization to blend the estimator together with the heuristic resolution.

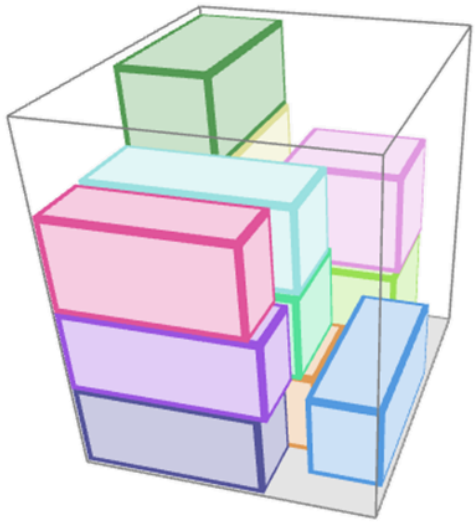

Figure 4.7: Graphical representation of the solution of the 3D-BPP.

4.2.2 The approach

For the optimization process under consideration, we are given the complete list of the problem specifics and are provided with realistic instances for validating it. Unfortunately, the number of instances is small compared to what is generally required for a data-driven model to reach a good level of approximation; to overcome the issue, we will resort to a data generation process that enables us to sample from an estimated distribution of the instance attributes.

In particular, we address the problem by dividing it into the following steps:

- *Distribution Inference*: the available instances are used to estimate the distribution of each attribute, e.g. the number of items to load or the physical attributes of the item (weight, length, etc).

- *Data Generation*: starting from the distributions inferred at the previous point, new instances are generated via standard sampling and a significant set of synthetic instances $\mathcal{X}$ is constructed.

- *Optimization*: for each synthetic instance $x \in \mathcal{X}$, the corresponding optimization problem is solved to produce a labelled pair (x, y) that will

be added to the data set $\mathcal{D}$ that we represent, with a little notation abuse, as $\mathcal{D} = \{(x_i, y_i)\} = (\mathcal{X}, \mathcal{Y})$.

- *Learning*: once we have a proper data set $\mathcal{D}$ for training ML models, we run benchmark tests to identify the algorithms that perform best in inferring the number of bins from the instance attributes.

- *Integration*: we devise two different strategies to integrate the learned model within the heuristic resolution process.

Distribution Inference We are given a total of ca. 650 instances involving real sets of items. Unfortunately, most of the available instances are used by the company as test cases to check the correctness of the algorithm and, for this reason, they are not representative of the real-world scenarios we are interested in. Figure 4.8 represents the bin distribution over the input instances: the distribution is condensed around a small number of bins (in fact, around 80% of the input examples have 5 bins or less), while realistic instances usually involve tens of bins (a single airplane can load up to 8 bins). In order to overcome the issue, we resort to a data generation process, intended to provide a more realistic data set that will be used to train a ML model: more specifically, instead of using the input instances to train our learning model, we exploit them to infer how item features are distributed. The idea is to generate new instances with very low effort once we are able to sample items from a realistic item distribution. In fact, it seems reasonable to assume that each instance is characterized by the number and features of the items it contains, rather than the items themselves.
Furthermore, we make the strong assumption that item features are independent of each other, except of course for the computed ones (e.g. volume is the product of the dimensions of an item). Although this hypothesis may seem very strict, it also enables the verification of the optimization problem over a wide variety of item specifics; moreover, dimensional features such as length and width are likely to have no correlation with each other. There might be physical limitations on joint features, for example, volume and weight should be bounded for the density to be realistic, however, such items are very unlikely to be generated.

We start by collecting all the input items in a single item set $\mathcal{I}$; an item $i \in \mathcal{I}$ is characterized by a set of features $\mathcal{F}$, where we distinguish between numerical features $\mathcal{F}^r$ and categorical features $\mathcal{F}^c$. The inference procedure uses the

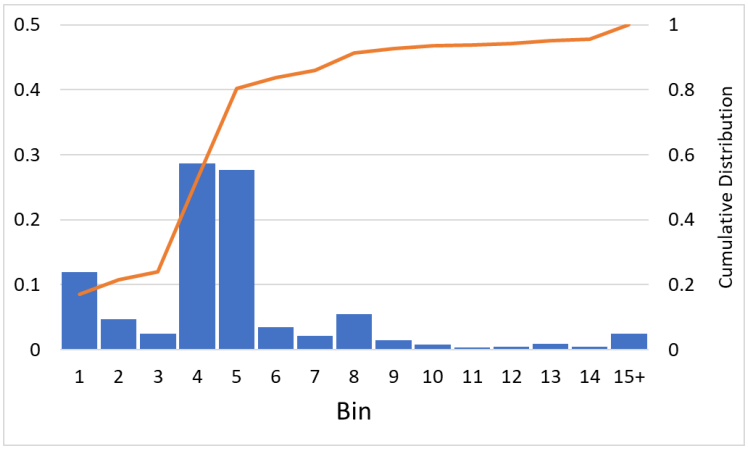

Figure 4.8: Distribution of the number of bins (histogram, left axis) and associated cumulative distribution (line, right axis) in the set of input instances. Note the double y-axis.

K-Medoids clustering algorithm [KR09] to group the initial item set $\mathcal{I}$ into K cluster $\{\mathcal{I}_1, \ldots, \mathcal{I}_K\}$ and identify the *seeds* s_k, $k = 1, \ldots, K$, that coincide with the cluster medoids. The seeds are regarded as the items most representative of the associated item population and will be the starting points for the generation of synthetic data. The dissimilarity measure used for the clustering process is the Euclidean distance between the item features after they have been standardized.

Each cluster is appointed with different feature distributions: in this way, each item seed will be enhanced with additional data reflecting the distribution of its neighbors. In practice, for each item feature $f_j \in \mathcal{F}$ we introduce a parametric distribution that we infer for each cluster k via regression on the corresponding data; we express the probability distribution as $P_k(f_j)$. We employ two näive but practical distributions: numerical features are represented by continuous uniform distributions, $f_j^{(r)} \sim Uni(a, b)$, whose extremes correspond to the maximum and minimum value of the feature over the considered cluster, while discrete features are described by categorical distributions with probabilities equal to the frequency of the corresponding value.

Name	Categorical	Numerical
height		✓
width		✓
length		✓
rotable	✓	
heavy	✓	
weight		✓
volume		✓
needStability	✓	
needRopes	✓	
riskCategory	✓	

Table 4.1: List of features characterizing an item of the 3D-BPP instance and relative type.

$$s_k = \{(f_j, P_k(f_j)) \ \ \forall f_j \in \mathcal{F}\} \quad \forall k \in K \tag{4.10}$$

$$P_k(f_j) = \begin{cases} Uni\left(\min_{\mathcal{I}_k} f_j, \max_{\mathcal{I}_k} f_j\right) & if \ f_j \in \mathcal{F}^{(r)} \\ Cat\left(f_j\right) & if \ f_j \in \mathcal{F}^{(c)} \end{cases} \tag{4.11}$$

Data Generation Once the distribution of each item feature has been estimated, synthetic items can be generated from Equation (4.11). Because we assume that the features are independent of each other, the total feature distribution is simply given by the product of the single distributions.

$$P(f_1, \ldots, f_n) = \prod_{j=1}^{N} P(f_j) \tag{4.12}$$

We generate a synthetic instance as follows: we first sample a random number of items n comprised between 50 and 150, $n \simeq Uni(50, 150)$; then we have to decide upon the distribution of the items over the seeds s_k. We extract random vector of positive numbers $m = (m_1, \ldots, m_K)$, $\|m\|_1 = 1$, that will define the approximate mixture of seeds in the resulting item set. The seeds are used uniquely as the starting point of the generation process (in practice, they are needed to encode constant parameters): the generation of a synthetic item from a seed s_k corresponds to sampling each feature value $f_j \in \mathcal{F}$ according to the distribution $P_k(f_j)$.

Optimization The 3D-BPP [MPV00] is a generalization of the well-known Bin Packing Problem and results strongly NP-hard. Exact resolution methods exist, for instance exploiting Branch&Bound method on a decomposed two-stage version of the original problem [MPV00], however, they usually involve a huge computational effort. Such computational burden is often made even bigger by the presence of additional constraints, for instance, balance constraints, item stacking constraints, or ad-hoc limitations on how certain items can be placed within the bin (e.g. an item may need ropes for stability, or it may or may not be rotated before loading). Given the complexity of the problem, many heuristic approaches have been developed: Guided Local Search [FPZ03], Tabu Search [LMV04; CPT09], Biased Randomized Key Genetic Algorithm [GR13], Extreme Point Heuristics [CPT08], Greedy Randomized Adaptive Search Procedure [Par+10]. Such approaches have the advantage of finding good solutions in a reasonable time, making the problem of real use in practical scenarios.
In our case, the resolution process is based on a Ruin&Recreate heuristic algorithm, a constructive approach that alternates different heuristic methods. The solution is progressively constructed from scratch by means of creation operators; once a complete solution is found, it is (partially) deconstructed, removing the less promising element-bin assignments, to be reconstructed again at the following iteration. In fact, this process allows to explore the feasible space by means of stacked moves. The choice of which operator to use is entrusted to a probability vector, which is dynamically updated and keeps track of each operator performance, favoring the ones that produced better solutions in past iterations.

For the scope at stake, we will not go into the detailed description of the problem constraints, nor give the exact implementation of the operators, but rather provide a high-level description of the resolution process, as it is needed to understand the integration methods that will be introduced later on. To some extent, we consider the solving process as a black-box and use the bound estimator to guide its exploration over the feasible space.
Among the creation operators we have:

- *LayerFiller operator*: selects the extreme point with the largest residual space and tries to fill it with available items.
- *BestFit operator*: given an item $i \in \mathcal{I}$, finds the best available position among the existing bins.

- *RandomizedBestFit operator*: it is the randomized version of the *BestFit operator*; for a given item, the best positions are evaluated and ranked according to a fitness function and one among them is randomly selected, with a probability corresponding to the position (inverse) ranking.

- *SSBestFit operator*: acts in a way similar to the *BestFit operator*, but randomly selects a rotation for all the items of the same size, before placing them in the bins.

On the other hand, we have a single destruction operator that clears the B bins having the biggest unused volume. The number B amounts approximately to one third of the current bins.

Given an instance, corresponding to a set of items $\mathcal{I}$, we denote the set of feasible solutions with $\mathcal{S}(\mathcal{I}) = \{S_m(\mathcal{I})\}$, $m = 1, \ldots, M$ and with $\hat{\mathcal{S}}(\mathcal{I})$ the incomplete solutions, corresponding to partial assignments of items to bins.

A creation operator $\mathcal{C}_i$, $i = 1, \ldots, C$ takes as arguments a (partial) solution and a set of items and outputs a complete solution

$$\mathcal{C}_i : \hat{S}_m(\mathcal{I}) \to S_m(\mathcal{I}) \tag{4.13}$$

A destruction operator $\mathcal{D}_i$ acts on a (partial) solution and eliminates some assignments of items to bins

$$\mathcal{D}_i : \hat{S}_m(\mathcal{I}) \to \hat{S}_m(\mathcal{I}) \tag{4.14}$$

Within the formalism introduced in Section 1.3, both operators can be represented as *refine* transitions acting on the current model state, $\mathcal{M} = \langle X, f, g \rangle$. The heuristic resolution algorithm is represented by a resolution state, initialized as $\mathcal{R} = \langle \mathcal{M}, \emptyset, \emptyset \rangle$, solved by means of the usual $\underset{s}{\to}\underset{g}{\to}\underset{z}{\to}$ transitions

$$\langle \mathcal{M}, \emptyset, \emptyset \rangle \underset{s}{\to} \langle \emptyset, \mathcal{M}, \emptyset \rangle \underset{g}{\to} \langle \mathcal{M}', \emptyset, \emptyset \rangle \underset{z}{\to} \langle \mathcal{M}', \emptyset, z(\mathcal{M}) \rangle$$

where the generating function g coincides with the application of destruction and a creation operator, as described in Equations (4.13) and (4.14). The idea behind this algorithm is that creation operators act in different ways and lead to distinct feasible solutions, thus allowing to explore various regions of the feasible space. Notice we have two extreme cases, corresponding to a complete

Algorithm 2 RUIN & RECREATE

input item set $\mathcal{I}$, problem parameters, operators $\mathcal{C}_i$ and $\mathcal{D}_i$, probabilities p

Initialize iteration counter $k = 0$, empty solution $S = \emptyset$, set of solutions $\mathcal{S} = \{\emptyset\}$

while stopping criterion not met **do**

$\hat{S}^{k+1} \leftarrow \mathcal{D}(S^k)$ ▷ ruin operator

$\mathcal{C} \leftarrow Choose(\mathcal{C}_i; p^k)$

$S^{k+1} \leftarrow \mathcal{C}(\hat{S}^{k+1})$ ▷ create operator

$\mathcal{S} \leftarrow \mathcal{S} \cup S^{k+1}$

$Update(p^k; S^{k+1}, S^k)$ ▷ operator probabilities updated

$k \leftarrow k + 1$

end while

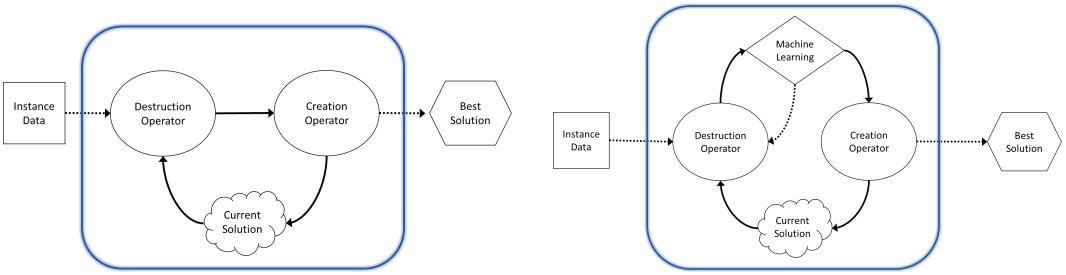

Figure 4.9: Bound estimation problem: graphical representation of the resolution process in the unbounded (left) and bounded (right) cases.

solution and the empty solution, for the creation and destruction operators, respectively. In fact, $\mathcal{C}_i(S(\mathcal{I})) = S(\mathcal{I})$ and $\mathcal{D}_i(\emptyset) = \emptyset$.
The resolution process is described in Algorithm 2 and depicted in Figure 4.9 (left); it is an iterative process that refines the solution until a stopping criterion is met, usually expressed in terms of the number of iterations. The iteration number corresponds to the creation operations (the number of creation and destruction operations may not coincide).

Learning The goal of the learning model is to estimate the number of bins y_i needed to load all the items of the input instance $\mathcal{I}$; because the instance consists of a collection of items and thus has a variable size, we first need to transform data in a representation acceptable by the ML models. Standard models work with input of fixed size, that requires us to compress each instance in a fixed number of input features: for numerical features, we use the first two

order moments obtained by the feature values, i.e. mean and variance, and the total sum, to account for extensive quantities, while categorical features are transformed via standard One-Hot Encoding. An alternative approach would be to use models working on input of variable size, for instance, Recurrent Neural Network and architectures alike; we do not implement such an approach for now.
The features used are the ones described in Table 4.1, augmented with the following ones: the number of items in the instance, the *compactness* of each item, and two binary features to distinguish heavy or long items.

Integration We devise three bounding strategies, with the last being a sanity check of the formers:

1. *Bounded Search* (S1): after each ruin operation, the learned bound estimator is used to evaluate whether the actual number of bins used can be improved or not, i.e. if the number of bins required to pack the items that have just been removed could be smaller than the current. If this is the case, we proceed with a create operator to build a complete solution; on the contrary, the ruin operator will be applied (see Algorithm 3). More specifically, given a complete solution $S(\mathcal{I})$ with an associated number of bins $n(S)$, the ruin operator produces a partial solution and an item set $\hat{S}(\mathcal{I})$ and $\hat{\mathcal{I}}$ such that $\hat{\mathcal{I}}$ encodes the unassigned items of $\hat{S}$; the latter becomes the input of the bound estimator $l_b(\hat{\mathcal{I}}) = \hat{n}$ and we have an estimated improvement when $n(\hat{S}) + \hat{n} < n(S)$.

2. *Global bound* (S2): at the beginning of the resolution process, we use the learner l_b to infer the number of bins required to pack the item set $\mathcal{I}$, $\hat{n}_{\mathcal{I}} = l_b(\mathcal{I})$. Then, the resolution process proceeds as always, but we stop as soon as the current solution has a number of bins equal to or smaller than the estimated bound.

3. *Random Search* (S3): in order to have a näive benchmark for comparison, a third strategy is devised: it is specular to the *Guided Search* one, except that the decision upon proceeding to the creation or not is now taken at random. This provides a baseline for comparing the strategies based on learning methods.

Algorithm 3 Ruin & Recreate With ML Bounding

input item set $\mathcal{I}$, problem parameters, operators $\mathcal{C}_i$ and $\mathcal{D}_i$, probabilities p, model l_b

```
Initialize iteration counter k = 0, empty solution S = ∅, set of solutions 𝒮 = {∅}
while stopping criterion not met do
    Ŝ^{k+1} ← 𝒟(S^k)                          ▷ ruin operator
    n̂ ← l_b(Ŝ^{k+1})                          ▷ bound estimation
    if n̂ < n(S^k) or Ŝ^{k+1} = ∅ then
        𝒞 ← Choose(𝒞_i; p^k)
        S^{k+1} ← 𝒞(Ŝ^{k+1})                  ▷ create operator
        𝒮 ← 𝒮 ∪ S^{k+1}
        Update(p^k; S^{k+1}, S^k)             ▷ operator probabilities updated
    end if
    k ← k + 1
end while
```

4.2.3 Experimental Results

In Figure 4.10 are reported the scores of different models on synthetically generated instances: we constructed 2000 synthetic instances of varying sizes and use an 80-20 split for the train and test set. Learning models are validated with a 5-fold cross-validation (with a fixed seed), while test instances are kept for evaluating the bounding strategies. The metric used for comparison is the Mean Absolute Error

$$MAE(y, \hat{y}) = \frac{1}{N} \sum_{i=1}^{N} |y_i - \hat{y}_i|$$

The models used are the following: Linear Regressor (LR), Random Forest Regressor (RFR), Extreme Gradient Boost Regressor (XGB), Support Vector Regressor (SVR), K-Nearest Neighbor Regressor (KNN), and a Fully-Connected Deep Neural Network (DNN). Except for XGB and DNN, models are taken from SciKit-Learn library [Ped+11] and implemented with standard parameters; XGB is taken from [CG16] and the DNN is a fully connected feed-forward network with three layers of respectively (32, 32, 16) neurons and *ReLU* activation function. Notice that all the above-mentioned models are regressors, meaning their output is a real value, while the number of bins is discrete by def-

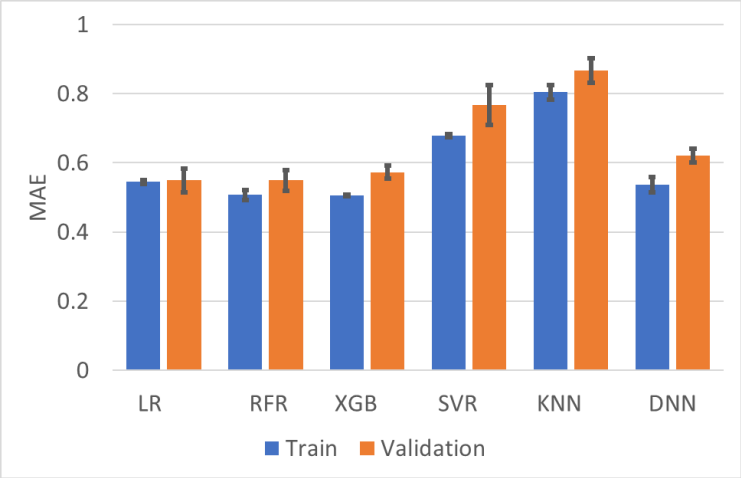

Figure 4.10: Mean Absolute Error for bin estimation on the train and validation sets for different ML models. Mean value and error bars are obtained after a 5-fold Cross-Validation procedure.

inition. Therefore, *the output is trimmed by rounding it to the biggest integer value*, i.e. with a ceiling operation $\hat{y}_i = \lceil f(x_i; \theta) \rceil$.

The learning models show similar performances both on train and validation data, with an error estimate of ca. half bin and a limited over-fitting. In particular, the LR model shows good performance despite its reduced complexity: this is mostly due to the fact that the number of bins has a very high linear correlation with extensive quantities such as the total volume of the items to be packed. Moreover, extensive and intensive quantities result quite dissimilar in behavior, with the former containing the most useful information, as can be seen from Figure 4.11.

The performances of the ML models appear more than acceptable. However, since we want to devise a bounding mechanism, we are also interested in the distribution of the error, in particular, we want to study whether our estimator tends to overestimate or underestimate the target distribution. This is shown in Figure 4.12: the error distribution is very similar among the learning models used. The error is computed as $Err_i = y_i - \hat{y}_i$ and results are biased towards positive numbers, meaning the models tend to underestimate the true number of bins. We can draw two considerations from this: 1) the bias is quite unexpected since the ceiling operation that trims the model's continuous output

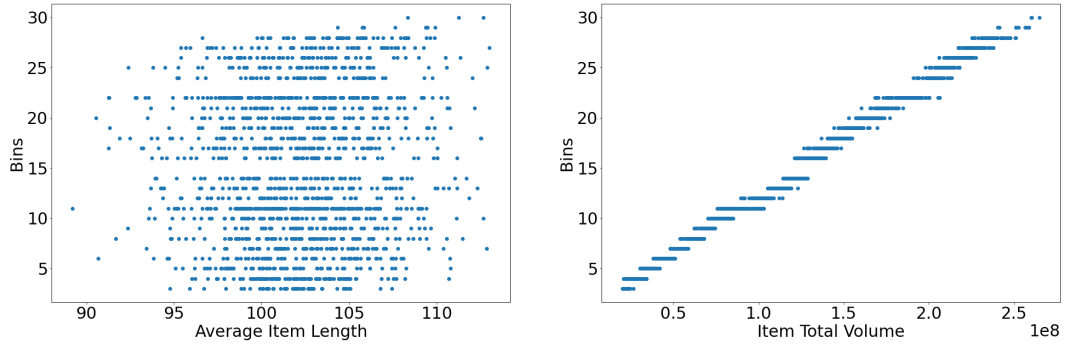

Figure 4.11: Distribution of intensive (left) and extensive (right) features with respect to the number of bins.

should pull towards an overestimation of the number of bins; 2) since we are interested in a bounding mechanism, it is preferable an underestimate of the number of bins rather than the contrary.
For the bounding tests, we select the two ML models that have the best validation error and also result more suited in limiting the overfitting, that are the Linear Regressor and the Random Forest Regressor models.

The experimentation is carried out as follows: for each bounding strategy and learning model, we optimize the 400 test instances with and without the bounding mechanism. For each test instance, we keep track of the final number of bins and the iterations needed to reach the best solution; we did the same experimentation setting the maximum number of iterations to 1000 and 10000, respectively. In Table 4.2 we report the results of the bounding experiments: each row represents a different pair of ML models and bounding strategy. We report the percentage of instances in which the bounded (B) (or unbounded (UB)) method resulted in better performances w.r.t the unbounded (or bounded), i.e. the solution has a smaller number of bins or, being equal to the number of bins, the number of iterations needed to reach the best solution is smaller. For example, the first row of the lower table corresponds to the bounding experiment using Linear Regressor, strategy S1 and 10000 iterations: in this case, out of the 400 test instances, the bounding strategy resulted in a solution with fewer number of bins 6.5% of the time. With an equal number of bins, the bounding strategy resulted in a fewer number of iterations to reach the solution in 10.8% of the cases.
The outcome of the bounding experiment is not satisfactory nor completely

Max Iterations = 1000

Model	Strategy	Bin Number		Best Iteration	
		UB	B	UB	B
LR	S1	3	9	85.5	0
	S2	91.5	1.3	4.2	0
	S3	17.8	22.5	59.7	0
RFR	S1	3	9	85	0
	S2	91.5	1.3	4.3	0
	S3	20	21	59	0

Max Iterations = 10000

Model	Strategy	Bin Number		Best Iteration	
		UB	B	UB	B
LR	S1	4.5	6.5	7.8	10.8
	S2	93.5	1.3	3	2.3
	S3	54	8	24	13.8
RFR	S1	4.5	7.5	11.5	13
	S2	93.5	1.3	3	2.3
	S3	55.5	7.5	22	15

Table 4.2: Results of bounding experiments: each row corresponds to a different bounding strategy. We report the percentage of instances in which the bounded method (B) and unbounded method (UB) resulted in an improvement for the number of bins or best iteration. The two tables correspond to different number of iterations in the heuristic algorithm.

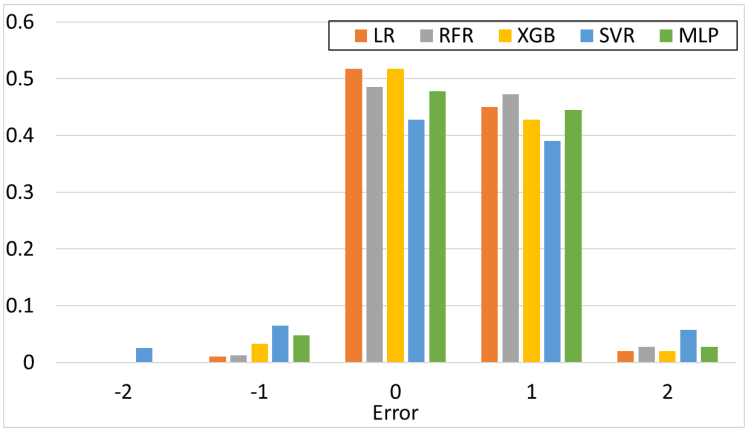

Figure 4.12: Error distribution for bin estimation for different learning models. The error corresponds to the difference between the true number of bins and the estimated one, $y_i - \hat{y}_i$.

clear: while the 'bounded search' (strategy S1) sometimes improves over the unbounded solution, its success is not robust. The global bound strategy (S2), on the other hand, has very poor results in the bin estimation, in spite of the prediction results seen before; this heavily affects the optimization results. The third strategy was added as a control experiment and confirms that by making random choices, the performance of the algorithm do not improve. This confirmation slightly ameliorates the results of bounded search (S1). The RFR model shows performances in line with the LR model and its results are consistent with the previous interpretations. The best results are obtained when the maximum iteration number is smaller: when the exploration is limited, it benefits more from the learning model. Conversely, a long random search corresponds to a thorough exploration, nullifying the positive impact of the bound estimator.

4.3 Conclusion

In this Chapter two applications have been presented: in the first one, a data-driven model is encoded within a COP. Given the simplicity of the prediction task and the structure of the COP, a lookup-table approach was preferred; however, the prediction rule $\lambda^{(i)}(p)$ of Equation (4.4) could be estimated through a learning model, as seen in Section 2.2.1 (for instance, encoding a tree-based

method using EML). The second experiment exhibits several issues that are frequently encountered when dealing with private companies and proposes a methodological approach: the scarcity of data is partially overcome with a data generation procedure, the optimization problem is tackled with an heuristic approach and a learning model is used to quickly estimate major KPIs of a solution without solving it. Furthermore, the learning model is exploited to boost the resolution process by either guiding heuristically the search or to define a bound on the objective function. Unfortunately, in spite of the promising results obtained in the predictive task, the integration experiments we performed did not give the expected improvements in performance; the best strategy is the one using the bin estimation to guide the search process, although it doesn't show sound result

Chapter 5

Conclusions and Future Directions

In this thesis, we have investigated the state of the art in the field of integration between Machine Learning and Constrained Optimization. The field is recent and still lacks a solid background, both from a theoretical and experimental point of view but results still very promising; we surveyed many of the latest works that we deemed to contain the most innovative contribution to the state of the art and we tried to classify them according to their integration scope.

We then introduced Moving Targets, a novel supervised training procedure to inject hard constraints into generic predictive models: the method, although relatively simple, shows good performances on the data sets and constraints considered and can be added to the arsenal of methods able to provide Machine Learning models with constraint support. Such techniques can highly improve the learning process both in terms of versatility and control. Therefore, Moving Targets constitutes a step forward in the direction of a more human-oriented AI, as it is general and allows for the realization of learning models that satisfy specific user requirements. This is a fundamental and essential behavior in contexts where there exist external constraints, which the trained AI model has to acknowledge.

Furthermore, we presented two real-world applications that integrate data-

driven models within a Combinatorial Optimization problem, either in the modeling or in the solving phase. In the first case, we leverage on learning a model to infer the failure probability of a network component from the historical track of failures. The model is directly encoded in the optimization problem and thus constitutes an example of full integration between the two fields. In the second scenario, a learning mechanism is employed for inferring the characteristics of a COP solution, without solving it; this aims at boosting the resolution process by providing a bounding mechanism. Unfortunately, results show that the integration is perhaps too näive and requires more thorough crafting.

The methods presented in this thesis cover different situations, both for the type of learning algorithm and optimization problem, and the interplay between the two of them. However, getting back to what was discussed in Chapter 2, techniques to integrate learning methods in Constrained Optimization Problems are affected by some common traits:

- **Modelling Hints**: learning models can be used to extract inner representations that can be used to gain insights on the problem structure of a combinatorial optimization problem.
- **Generalization**: devising a proper test set to evaluate the performance of the learned model on optimization instances requires extra carefulness and yet does not guarantee the same performances will be preserved during the optimization process.
- **Optimality**: optimality, as well as feasibility, of solutions generated by learning algorithms, is usually impossible to guarantee from a theoretical point of view. This is especially true for methods employing ML to produce complete solutions for the COPs.
- **Scaling**: learning models show high computational demand for training, both in terms of computational power and resources needed to generate the instances. However, their generalization capability may not compensate for the effort.

On the other side, learning models can hugely benefit from the theory and algorithms of Constrained/Combinatorial Optimization, both to boost model performances and to provide novel learning schemes, such as the one we presented in Chapter 3.

Bibliography

[AAV19] Sina Aghaei, Mohammad Javad Azizi, and Phebe Vayanos. "Learning optimal and fair decision trees for non-discriminative decision-making". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 33. 2019, pp. 1418–1426.

[AC18] Francisco J Aragón Artacho and Rubén Campoy. "A new projection method for finding the closest point in the intersection of convex sets". In: *Computational optimization and applications* 69.1 (2018), pp. 99–132.

[ALW17] Alejandro Marcos Alvarez, Quentin Louveaux, and Louis Wehenkel. "A machine learning-based approximation of strong branching". In: *INFORMS Journal on Computing* 29.1 (2017), pp. 185–195.

[AS19] Florian Arnold and Kenneth Sörensen. "Knowledge-guided local search for the vehicle routing problem". In: *Computers & Operations Research* 105 (2019), pp. 32 –46. ISSN: 0305-0548. DOI: https://doi.org/10.1016/j.cor.2019.01.002. URL: http://www.sciencedirect.com/science/article/pii/S0305054819300024.

[BA04] J Martin Bland and Douglas G Altman. "The logrank test". In: *Bmj* 328.7447 (2004), p. 1073.

[Bal+18] Maria-Florina Balcan et al. "Learning to Branch". In: *International Conference on Machine Learning*. 2018, pp. 344–353.

[BB10] Roberto Battiti and Mauro Brunato. "Reactive Search Optimization: Learning While Optimizing". In: *Handbook of Metaheuristics*. Ed. by Michel Gendreau and Jean-Yves Potvin. Boston, MA: Springer US, 2010, pp. 543–571. ISBN: 978-1-4419-1665-5. DOI: 10.1007/978-1-4419-1665-5_18. URL: https://doi.org/10.1007/978-1-4419-1665-5_18.

[BBV04] Stephen Boyd, Stephen P Boyd, and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[BD+03] Stephen Boyd, Jon Dattorro, et al. "Alternating projections". In: *EE392o, Stanford University* (2003).

[Bel+16] Irwan Bello et al. "Neural combinatorial optimization with reinforcement learning". In: *arXiv preprint arXiv:1611.09940* (2016).

[Ben97] Yoshua Bengio. "Using a financial training criterion rather than a prediction criterion". In: *International Journal of Neural Systems* 8.04 (1997), pp. 433–443.

[Ber+17] Richard Berk et al. "A Convex Framework for Fair Regression". In: *CoRR* abs/1706.02409 (2017). arXiv: 1706.02409. URL: http://arxiv.org/abs/1706.02409.

[Ber75] Dimitri P Bertsekas. "Necessary and sufficient conditions for a penalty method to be exact". In: *Mathematical programming* 9.1 (1975), pp. 87–99.

[Bes+13] Christian Bessiere et al. "Constraint acquisition via partial queries". In: *Twenty-Third International Joint Conference on Artificial Intelligence*. 2013.

[Bes+17] Christian Bessiere et al. "Constraint acquisition". In: *Artificial Intelligence* 244 (2017), pp. 315–342.

[Bis07] Christopher M. Bishop. *Pattern recognition and machine learning, 5th Edition*. Information science and statistics. Springer, 2007. ISBN: 9780387310732. URL: https://www.worldcat.org/oclc/71008143.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT press, 2008.

[BL+18] Radu Baltean-Lugojan et al. "Selecting cutting planes for quadratic semidefinite outer-approximation via trained neural networks". In: (2018).

[BLM15] Alessio Bonfietti, Michele Lombardi, and Michela Milano. "Embedding decision trees and random forests in constraint programming". In: *International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems*. Springer. 2015, pp. 74–90.

[BLZ18] Pierre Bonami, Andrea Lodi, and Giulia Zarpellon. "Learning a classification of mixed-integer quadratic programming problems". In: *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research*. Springer. 2018, pp. 595–604.

[Bot10] Léon Bottou. "Large-scale machine learning with stochastic gradient descent". In: *Proceedings of COMPSTAT'2010*. Springer, 2010, pp. 177–186.

[BS12] Nicolas Beldiceanu and Helmut Simonis. "A model seeker: Extracting global constraint models from positive examples". In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2012, pp. 141–157.

[Cal+17] Laura Calvet et al. "Learnheuristics: hybridizing metaheuristics with machine learning for optimization with dynamic inputs". In: *Open Mathematics* 15.1 (2017), pp. 261 –280. URL: `https://www.degruyter.com/view/journals/math/15/1/article-p261.xml`.

[Cap+21a] Quentin Cappart et al. "Combinatorial optimization and reasoning with graph neural networks". In: *arXiv preprint arXiv:2102.09544* (2021).

[Cap+21b] Quentin Cappart et al. "Combining Reinforcement Learning and Constraint Programming for Combinatorial Optimization". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 35. 5. 2021, pp. 3677–3687.

[CG16] Tianqi Chen and Carlos Guestrin. "Xgboost: A scalable tree boosting system". In: *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. 2016, pp. 785–794.

[Che+19] Di Chen et al. "Task-Based Learning via Task-Oriented Prediction Network". In: *arXiv preprint arXiv:1910.09357* (2019).

[Chr11] Symeon E Christodoulou. "Water network assessment and reliability analysis by use of survival analysis". In: *Water Resources Management* 25.4 (2011), pp. 1229–1238.

[Cir+20] Gabriele Ciravegna et al. "A constraint-based approach to learning and explanation". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 34. 04. 2020, pp. 3658–3665.

[Cot+19] Andrew Cotter et al. "Optimization with Non-Differentiable Constraints with Applications to Fairness, Recall, Churn, and Other Goals." In: *Journal of Machine Learning Research* 20.172 (2019), pp. 1–59.

[CPT08] Teodor Gabriel Crainic, Guido Perboli, and Roberto Tadei. "Extreme point-based heuristics for three-dimensional bin packing". In: *Informs Journal on computing* 20.3 (2008), pp. 368–384.

[CPT09] Teodor Gabriel Crainic, Guido Perboli, and Roberto Tadei. "TS2PACK: A two-level tabu search for the three-dimensional bin packing problem". In: *European Journal of Operational Research* 195.3 (2009), pp. 744–760.

[CSM14] Alison Cozad, Nikolaos V Sahinidis, and David C Miller. "Learning surrogate models for simulation-based optimization". In: *AIChE Journal* 60.6 (2014), pp. 2211–2227.

[CV95] Corinna Cortes and Vladimir Vapnik. "Support-vector networks". In: *Machine learning* 20.3 (1995), pp. 273–297.

[Dai+17] Hanjun Dai et al. *Learning Combinatorial Optimization Algorithms over Graphs*. 2017. arXiv: `1704.01665 [cs.LG]`.

[DAK17] Priya Donti, Brandon Amos, and J Zico Kolter. "Task-based end-to-end model learning in stochastic optimization". In: *Advances in Neural Information Processing Systems*. 2017, pp. 5484–5494.

[Dan51] George B Dantzig. "Maximization of a linear function of variables subject to linear inequalities". In: *Activity analysis of production and allocation* 13 (1951), pp. 339–347.

[Dan90] George B Dantzig. "Origins of the simplex method". In: *A history of scientific computing*. 1990, pp. 141–151.

[DDS16] Hanjun Dai, Bo Dai, and Le Song. "Discriminative embeddings of latent variable models for structured data". In: *International conference on machine learning*. 2016, pp. 2702–2711.

[Deu+18] Michel Deudon et al. "Learning heuristics for the tsp by policy gradient". In: *International conference on the integration of constraint programming, artificial intelligence, and operations research*. Springer. 2018, pp. 170–181.

[DG17] Dheeru Dua and Casey Graff. *UCI Machine Learning Repository*. Accessed on 15/01/20. 2017. URL: `http://archive.ics.uci.edu/ml`.

[DGS17] Michelangelo Diligenti, Marco Gori, and Claudio Sacca. "Semantic-based regularization for learning and inference". In: *Artificial Intelligence* 244 (2017), pp. 143–165.

[DLM21] Fabrizio Detassis, Michele Lombardi, and Michela Milano. "Teaching the Old Dog New Tricks: Supervised Learning with Constraints". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 35. 5. 2021, pp. 3742–3749.

[Dro+20] Iddo Drori et al. "Learning to solve combinatorial optimization problems on real-world graphs in linear time". In: *2020 19th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE. 2020, pp. 19–24.

[DRPT18] Luc De Raedt, Andrea Passerini, and Stefano Teso. "Learning constraints from examples". In: *Thirty-Second AAAI Conference on Artificial Intelligence*. 2018.

[EG17] Adam N Elmachtoub and Paul Grigas. "Smart" predict, then optimize"". In: *arXiv preprint arXiv:1710.08005* (2017).

[Fer+19] Aaron Ferber et al. "MIPaaL: Mixed integer program as a layer". In: *arXiv preprint arXiv:1907.05912* (2019).

[FF19] Martina Fischetti and Marco Fraccaro. "Machine learning meets mathematical optimization to predict the optimal production of offshore wind parks". In: *Computers & Operations Research* 106 (2019), pp. 289–297.

[FJ18] Matteo Fischetti and Jason Jo. "Deep neural networks and mixed integer linear optimization". In: *Constraints* 23.3 (2018), pp. 296–309.

[FPZ03] Oluf Faroe, David Pisinger, and Martin Zachariasen. "Guided local search for the three-dimensional bin-packing problem". In: *Informs journal on computing* 15.3 (2003), pp. 267–283.

[Gas+19] Maxime Gasse et al. "Exact combinatorial optimization with graph convolutional neural networks". In: *Advances in Neural Information Processing Systems*. 2019, pp. 15554–15566.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[GEW06] Pierre Geurts, Damien Ernst, and Louis Wehenkel. "Extremely randomized trees". In: *Machine learning* 63.1 (2006), pp. 3–42.

[GR13] José Fernando Gonçalves and Mauricio G.C. Resende. "A biased random key genetic algorithm for 2D and 3D bin packing problems". In: *International Journal of Production Economics* 145.2 (2013), pp. 500–510. ISSN: 0925-5273. DOI: https://doi.org/10.1016/j.ijpe.2013.04.019. URL: https://www.sciencedirect.com/science/article/pii/S0925527313001837.

[HDIE14] He He, Hal Daume III, and Jason M Eisner. "Learning to search in branch and bound algorithms". In: *Advances in neural information processing systems*. 2014, pp. 3293–3301.

[HTT20] Andre Hottung, Shunji Tanaka, and Kevin Tierney. "Deep learning assisted heuristic tree search for the container pre-marshalling problem". In: *Computers & Operations Research* 113 (2020), p. 104781.

[JLB19] Chaitanya K Joshi, Thomas Laurent, and Xavier Bresson. "An efficient graph convolutional network technique for the travelling salesman problem". In: *arXiv preprint arXiv:1906.01227* (2019).

[JM94] Joxan Jaffar and Michael J Maher. "Constraint logic programming: A survey". In: *The journal of logic programming* 19 (1994), pp. 503–581.

[Joa06] Thorsten Joachims. "Training linear SVMs in linear time". In: *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2006, pp. 217–226.

[KB14] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[KC09] Faisal Kamiran and Toon Calders. "Classifying without discriminating". In: *2009 2nd International Conference on Computer, Control and Communication*. IEEE. 2009, pp. 1–6.

[KC12] Faisal Kamiran and Toon Calders. "Data preprocessing techniques for classification without discrimination". In: *Knowledge and Information Systems* 33.1 (2012), pp. 1–33.

[Kha+16] Elias Boutros Khalil et al. "Learning to Branch in Mixed Integer Programming." In: *AAAI*. 2016, pp. 724–731.

[Kha+17] Elias B Khalil et al. "Learning to Run Heuristics in Tree Search." In: *IJCAI*. 2017, pp. 659–666.

[KLP17] Markus Kruber, Marco E Lübbecke, and Axel Parmentier. "Learning when to use a decomposition". In: *International Conference on AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*. Springer. 2017, pp. 202–210.

[KM03] John P Klein and Melvin L Moeschberger. *Survival analysis: techniques for censored and truncated data*. Vol. 1230. Springer, 2003.

[KNS09] Fatma Kılınç Karzan, George L Nemhauser, and Martin WP Savelsbergh. "Information-based branching schemes for binary linear mixed integer problems". In: *Mathematical Programming Computation* 1.4 (2009), pp. 249–293.

[Kot16] Lars Kotthoff. "Algorithm selection for combinatorial search problems: A survey". In: *Data Mining and Constraint Programming*. Springer, 2016, pp. 149–190.

[KR09] Leonard Kaufman and Peter J Rousseeuw. *Finding groups in data: an introduction to cluster analysis*. Vol. 344. John Wiley & Sons, 2009.

[Kum+20] Mohit Kumar et al. "Learning MAX-SAT from contextual examples for combinatorial optimisation". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 34. 04. 2020, pp. 4493–4500.

[Kum+21] Mohit Kumar et al. "Learning Mixed-Integer Linear Programs from Contextual Examples". In: *arXiv preprint arXiv:2107.07136* (2021).

[KVHW18] Wouter Kool, Herke Van Hoof, and Max Welling. "Attention, learn to solve routing problems!" In: *arXiv preprint arXiv:1803.08475* (2018).

[Lal+10] Arnaud Lallouet et al. "On learning constraint problems". In: *Tools with Artificial Intelligence (ICTAI), 2010 22nd IEEE International Conference on*. Vol. 1. IEEE. 2010, pp. 45–52.

[LD60] A. H. Land and A. G. Doig. "An Automatic Method of Solving Discrete Programming Problems". In: *Econometrica* 28.3 (1960), pp. 497–520. ISSN: 00129682, 14680262. URL: http://www.jstor.org/stable/1910129.

[Lem+19] Henrique Lemos et al. "Graph colouring meets deep learning: Effective graph neural network models for combinatorial problems". In: *2019 IEEE 31st International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE. 2019, pp. 879–885.

[LG16] Michele Lombardi and Stefano Gualandi. "A lagrangian propagator for artificial neural networks in constraint programming". In: *Constraints* 21.4 (2016), pp. 435–462.

[LMB17] Michele Lombardi, Michela Milano, and Andrea Bartolini. "Empirical decision model learning". In: *Artificial Intelligence* 244 (2017). Combining Constraint Solving with Mining and Learning, pp. 343 –367. ISSN: 0004-3702. DOI: https://doi.org/10.1016/j.artint.2016.01.005. URL: http://www.sciencedirect.com/science/article/pii/S0004370216000126.

[LMV04] Andrea Lodi, Silvano Martello, and Daniele Vigo. "TSpack: a unified tabu search code for multi-dimensional bin packing problems". In: *Annals of Operations Research* 131.1 (2004), pp. 203–213.

[LRT11] Binh Thanh Luong, Salvatore Ruggieri, and Franco Turini. "k-NN as an implementation of situation testing for discrimination discovery and prevention". In: *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2011, pp. 502–510.

[LZ17] Andrea Lodi and Giulia Zarpellon. "On learning and branching: a survey". In: *Top* 25.2 (2017), pp. 207–236.

[Ma+19] Qiang Ma et al. "Combinatorial optimization by graph pointer networks and hierarchical reinforcement learning". In: *arXiv preprint arXiv:1911.04936* (2019).

[MAWL16] Alejandro Marcos Alvarez, Louis Wehenkel, and Quentin Louveaux. "Online learning for strong branching approximation in branch-and-bound". In: (2016).

[MPV00] Silvano Martello, David Pisinger, and Daniele Vigo. "The three-dimensional bin packing problem". In: *Operations research* 48.2 (2000), pp. 256–267.

[MR04] Michela Milano and Andrea Roli. "Magma: A multiagent architecture for metaheuristics". In: *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 34.2 (2004), pp. 925–941.

[MRT18] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of machine learning*. MIT press, 2018.

[Nai+20] Vinod Nair et al. "Solving mixed integer programs using neural networks". In: *arXiv preprint arXiv:2012.13349* (2020).

[Naz+18] MohammadReza Nazari et al. "Reinforcement Learning for Solving the Vehicle Routing Problem". In: *Advances in Neural Information Processing Systems* 31 (2018).

[Now+17] Alex Nowak et al. "A Note on Learning Algorithms for Quadratic Assignment with Graph Neural Networks". In: *CoRR* abs/1706.07450 (2017). arXiv: 1706.07450. URL: http://arxiv.org/abs/1706.07450.

[NW06] Jorge Nocedal and Stephen Wright. *Numerical optimization*. Springer Science & Business Media, 2006.

[Par+10] Francisco Parreño et al. "A hybrid GRASP/VND algorithm for two-and three-dimensional bin packing". In: *Annals of Operations Research* 179.1 (2010), pp. 203–220.

[Pau+21] Anselm Paulus et al. "CombOptNet: Fit the Right NP-Hard Problem by Learning Integer Programming Constraints". In: *arXiv preprint arXiv:2105.02343* (2021).

[PB+14] Neal Parikh, Stephen Boyd, et al. "Proximal algorithms". In: *Foundations and Trends® in Optimization* 1.3 (2014), pp. 127–239.

[Ped+11] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[PK17] Tomasz P Pawlak and Krzysztof Krawiec. "Automatic synthesis of constraints from examples using mixed integer linear programming". In: *European Journal of Operational Research* 261.3 (2017), pp. 1141–1157.

[Pou+18] Samira Pouyanfar et al. "A survey on deep learning: Algorithms, techniques, and applications". In: *ACM Computing Surveys (CSUR)* 51.5 (2018), pp. 1–36.

[Poz+21] Matteo Pozzi et al. "District heating network maintenance planning optimization". In: *Energy Reports* 7 (2021). The 17th International Symposium on District Heating and Cooling, pp. 184–192. ISSN: 2352-4847. DOI: https://doi.org/10.1016/j.egyr.2021.08.156. URL: https://www.sciencedirect.com/science/article/pii/S2352484721007605.

[Pra+18] Marcelo O. R. Prates et al. "Learning to Solve NP-Complete Problems: A Graph Neural Network for Decision TSP". In: *AAAI*. 2018.

[PS98] Christos H Papadimitriou and Kenneth Steiglitz. *Combinatorial optimization: algorithms and complexity*. Courier Corporation, 1998.

[Que+05] Nestor V Queipo et al. "Surrogate-based analysis and optimization". In: *Progress in aerospace sciences* 41.1 (2005), pp. 1–28.

[RVBW06] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming*. Elsevier, 2006.

[SB18] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[Sca+08] Franco Scarselli et al. "The graph neural network model". In: *IEEE transactions on neural networks* 20.1 (2008), pp. 61–80.

[Sel+18] Daniel Selsam et al. "Learning a SAT Solver from Single-Bit Supervision". In: *International Conference on Learning Representations*. 2018.

[SG20] Helge Spieker and Arnaud Gotlieb. "Learning Objective Boundaries for Constraint Optimization Problems". In: *International Conference on Machine Learning, Optimization, and Data Science*. Springer. 2020, pp. 394–408.

[SS+11] Shai Shalev-Shwartz et al. "Pegasos: Primal estimated sub-gradient solver for svm". In: *Mathematical programming* 127.1 (2011), pp. 3–30.

[STÖ19] Heda Song, Isaac Triguero, and Ender Özcan. "A review on the self and dual interactions between machine learning and optimisation". In: *Progress in Artificial Intelligence* 8.2 (2019), pp. 143–165. ISSN: 2192-6360. DOI: 10.1007/s13748-019-00185-z. URL: https://doi.org/10.1007/s13748-019-00185-z.

[Vel+17] Petar Veličković et al. "Graph attention networks". In: *arXiv preprint arXiv:1710.10903* (2017).

[VFJ15] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. "Pointer Networks". In: *Advances in Neural Information Processing Systems 28*. Ed. by C. Cortes et al. Curran Associates, Inc., 2015, pp. 2692–2700. URL: http://papers.nips.cc/paper/5866-pointer-networks.pdf.

[VG01] Rob J Van Glabbeek. "The linear time-branching time spectrum I. The semantics of concrete, sequential processes". In: *Handbook of process algebra*. Elsevier, 2001, pp. 3–99.

[Vu+17] Ky Khac Vu et al. "Surrogate-based methods for black-box optimization". In: *International Transactions in Operational Research* 24.3 (2017), pp. 393–424.

[WDT19] Bryan Wilder, Bistra Dilkina, and Milind Tambe. "Melding the data-decisions pipeline: Decision-focused learning for combinatorial optimization". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 33. 2019, pp. 1658–1665.

[XKK18] Hong Xu, Sven Koenig, and TK Satish Kumar. "Towards effective deep learning for constraint satisfaction problems". In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2018, pp. 588–597.

[XQA21] Álinson S Xavier, Feng Qiu, and Shabbir Ahmed. "Learning to solve large-scale security-constrained unit commitment problems". In: *INFORMS Journal on Computing* 33.2 (2021), pp. 739–756.

[Xu+19] Keyulu Xu et al. "What can neural networks reason about?" In: *arXiv preprint arXiv:1905.13211* (2019).

[YP19] Emre Yolcu and Barnabás Póczos. "Learning Local Search Heuristics for Boolean Satisfiability." In: *NeurIPS*. 2019, pp. 7990–8001.

[Zar+20] Giulia Zarpellon et al. "Parameterizing branch-and-bound search trees to learn branching policies". In: *arXiv preprint arXiv:2002.05120* (2020), p. 12.

[Zha+20] Hang Zhao et al. "Online 3D Bin Packing with Constrained Deep Reinforcement Learning". In: *arXiv preprint arXiv:2006.14978* (2020).

[Zhu+21] Qianwen Zhu et al. "Learning to Pack: A Data-Driven Tree Search Algorithm for Large-Scale 3D Bin Packing Problem". In: *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*. 2021, pp. 4393–4402.