ALMA MATER STUDIORUM · UNIVERSITY OF BOLOGNA

SCHOOL OF ENGINEERING AND ARCHITECTURE

DEPARTMENT of COMPUTER SCIENCE and ENGINEERING (DISI)

TWO-YEAR MASTER’S DEGREE in COMPUTER ENGINEERING

MASTER’S THESIS

In

Intelligent Systems M

Automated Configuration of Offline/Online Algorithms: an Empirical Model Learning Approach

Supervisor:

Prof. MILANO MICHELA

Candidate:

MINERVA MICHELA

Co-supervisors:

Dr. DE FILIPPO ALLEGRA

Dr. BORGHESI ANDREA

Academic Year 2019/2020

Session III

Abstract

The energy management system is the intelligent core of a virtual power plant and it manages power flows among units in the grid. This implies dealing with optimization under uncertainty because entities such as loads and renewable energy resources have stochastic behaviors. A hybrid offline/online optimization technique can be applied in such problems to ensure efficient online computation.

This work devises an approach that integrates machine learning and optimization models to perform automatic algorithm configuration. It is inserted as the top component in a two-level hierarchical optimization system for the VPP, with the goal of configuring the low-level offline/online optimizer.

Data from the low-level algorithm is used for training machine learning models - decision trees and neural networks – that capture the highly complex behavior of both the controlled VPP and the offline/online optimizer. Then, Empirical Model Learning is adopted to build the optimization problem, integrating usual mathematical programming and ML models.

The proposed approach successfully combines optimization and machine learning in a data-driven and flexible tool that performs automatic configuration and forecasting of the low-level algorithm for unseen input instances.

Table of Contents

List of Figures

List of Graphs

List of Tables

List of Acronyms

- **Adam**: Adaptive moment estimation.
- **ANN**: Artificial neural network.
- **DER**: Distributed energy resource.
- **DT**: Decision tree.
- **DS**: Decision stump.
- **EML**: Empirical Model Learning.
- **EMS**: Energy management system.
- **GD**: Gradient descent.
- **GRS**: Greedy recursive splitting.
- **KDE**: Kernel density estimate.
- **KNN**: K-nearest neighbors.
- **LASSO**: Least absolute shrinkage and selection operator.
- **LR**: Linear regression.
- **MAE**: Mean absolute error.
- **MILP**: mixed-integer linear programming-
- **ML**: Machine learning.
- **MSE**: Mean squared error.
- **NN**: Neural network.
- **nTraces**: Number of traces.
- **PCA**: Principal component analysis.
- **PV**: RES generation power.
- **ReLU**: Rectified linear unit.
- **RES**: Renewable energy source.
- **RBF**: Radial basis function
- **RF**: Random forest.
- **SGD**: Stochastic gradient descent.
- **SVM**: Support vector machine.
- **VPP**: Virtual power plant.

Introduction

Smart grids are the evolution of electrical grids, interconnected infrastructures that deliver electrical energy to consumers. They integrate novel power sources into a distributed energy resources scenario. Smart grids leverage state-of-the-art control and information techniques to perform monitoring, control, and forecasting on the complex network of distributed entities, named virtual power plant. The core of the VPP is the energy management system; it is the orchestrator of such a large system and it manages its power flows. By virtue of the EMS, a smart grid enables enhanced energy efficiency, flexibility, security, and reliability in the power distribution; it promotes green power sources thus helping the environment and it allows money savings by adopting smart energy management.

The EMS in a VPP decides power flows in the grid and it operates under a specific objective, usually the minimization of operational costs. This is a problem of optimization under uncertainty because a smart grid integrates elements with stochastic behavior, e.g. renewable energy resources and loads. Hybrid offline/online algorithms can be applied to perform online optimization, namely, to decide power flows in real-time given the real conditions of the system. They adopt hefty stochastic optimization algorithms but shift part of their computation offline to reduce online costs. An offline/online technique is based on the expensive offline computation of a contingency table; it contains information on possible online scenarios called traces. In the online step, a very efficient fixing heuristic makes decisions guided by the contingency table; it adjusts offline-computed solutions to actual conditions. The number of traces guiding the fixing heuristic is a fundamental parameter to configure; it balances a tradeoff between solution quality and computation cost.

This work devises an approach to perform automatic configuration of the offline/online algorithm for new unseen instances. It introduces a combinatorial optimization problem on top of the offline/online method, resulting in a two-levels hierarchical optimization system for the VPP. We want to define problem-specific objectives and constraints for the low-level optimization, for example concerning solution quality, online computation time, or resources. These are entailed in the high-level optimization problem. The proposed approach is used ahead of the online step to guide its design or get forecasts about its performance on unseen instances. It is an automatic tool for the configuration of the low-level algorithm that allows one to automatically decide or predict its parameters, run-time, or computational resources based on desired constraints and objectives.

The high-level optimizer captures the behavior of the controlled system, both the VPP and the hybrid offline/online algorithm. We leverage machine learning techniques, with a focus on decision trees and neural networks, to model the highly complex relationships between variables involved in the system.

The proposed approach leverages Empirical Model Learning to integrate empirical ML models and combinatorial optimization problems. Machine learning models that capture real-world relationships among variables are embedded in the optimization problem via EML; once encoded, they are used by the solver to generate solutions and boost the resolution process. The EML-based approach allows a flexible and completely data-driven design of the optimization model; it does not require specific knowledge about the VPP system or the hybrid optimization process, and it removes the need of a manually-crafted modeling phase by domain experts. EML brings together declarative models from optimization research and predictive models from machine learning, and it shows that optimization in complex real-world systems is possible.

We perform a preliminary study on machine learning techniques to assess how the relationships among variables are captured by different models. Then, we build decision trees and neural networks and we finally embed them in combinatorial optimization models. We compare optimization models based on DTs and NNs on several examples to shed light on their strengths and weaknesses. Finally, we apply different high-level optimizers, modeled using trees with distinct hyperparameters, on two real-world use cases.

Chapter 1
Virtual Power Plant

1.1 Smart Grid

Production, provisioning and consumption of electricity has been an important matter of research since the 18th century. The provisioning infrastructures currently used across the world to distribute power descend from the first alternating current (AC) system studied by Nikola Tesla with Westinghouse Electric in the late 1880s. The generation of electricity was localized around communities in a time when the energy demand was ridiculous compared to today, namely, few lightbulbs and power-alimented devices. A limited number of significant changes were made to the power distribution networks ever since. The "centralized top-down" power grid is designed for a unidirectional delivery of electricity to consumers. Power is generated in few large power plants, and from these locations it is distributed to customers through the power grid infrastructure. These systems are not designed for the continuously rising demand of the 21st century; the centralized design implies energy losses, in particular for long distances, together with significant construction and maintenance costs. Moreover, regular electrical grids do not meet the need of flexibility and they do not exploit the significant amount of data and resources available nowadays.

In recent years both production and consumption of energy have been advancing rapidly. Electrical grids have evolved with a progressive shift towards decentralized generation of energy. The development of several new energy resources allows production of sufficient amounts of energy to support the always-growing demand of energy. The introduction of green resources also meets the necessity of switching to environment friendly energy sources to reduce human's footprint on a growingly impacted Earth. Moreover, consumers' products and needs are changing in this direction: new smart sensors, devices, and smart appliances are available to consumers at large scale. An increasing number of products are becoming part of the electric networks; an important example of this phenomenon is the growing market of electric vehicles.

This evolution is happening in parallel with large changes in computational capabilities available to humans. Technologies like cloud computing and big data techniques allow generation, storing, and elaboration of massive amounts of data. In the meantime, Artificial Intelligence (AI) allows scientists to use these data for analyzing and forecasting tasks in several different applications.

Finding themselves in between these two worlds, *smart grids* represent the modern evolution of regular electrical grids and assumed a role of increasing importance in both industrial and academic research. According to the European Union Regulation 347/2013 [1], *"'Smart grid' means an electricity network that can integrate in a cost efficient manner the behavior and actions of all users connected to it, including generators, consumers and those that both generate and consume, in order to ensure an economically efficient and sustainable power system with low losses and high levels of quality, security of supply and safety".*

A smart grid integrates *distributed energy resources* (*DERs*), both conventional sources and new types such as renewable resources (*RESs*). DERs represent the most important factor for decentralized generation and consumption of energy. They can be generation systems - both renewable and non-renewable, such as wind and solar power plants, biomass plants, gas generators, and conventional energy generation sources -, energy storage systems (ESS) or loads - such as building loads -. DER elements that are peculiar in smart grids are *energy microgeneration entities*; for example, a building that produces, stores, and shares energy generated through renewable sources.

Smart grids enable the most recent technologies to be integrated in the power system, both for production and consumption. Some examples are green energy sources such as wind and solar energy units for production, or smart home devices and electric vehicles for consumption.

Smart grids overcome the mono-directionality of the old infrastructure. They introduce a two-way dialog where not only electricity but also information is exchanged between producers and customers. Data resulting from this communication is used by the systems for management purposes.

Smart grids use digital information, automation, and control technologies to increase energetic efficiency (namely, use less energy), provide flexibility and ensure security and reliability of the electric grid. They integrate smart technologies for monitoring and forecasting on all actors involved in the grid, from producers to smart appliances and devices. They leverage technology and data to increase economic efficiency for customers and to increase sustainability of the power system.

1.2 Virtual Power Plant

In a smart grid, several decentralized entities are connected and interact in a complex way. This sophisticated system must be orchestrated in order to achieve maximum efficiency while maintaining reliability. The network control structure must evolve to be able to handle distributed power resources, to ensure the flexibility requested to energy systems, to meet the needs of increasingly complex customer devices and to manage the variability of RESs. *virtual power plants* (*VPPs*) come into play in this important role.

A VPP is a distributed power plant; it aggregates and manages units connected to the electrical grid to produce, store, and use energy, allowing them to operate as a unified power plant. It clusters and orchestrates several little distributed energy resources, generating a more flexible and secure energy supply compared to a conventional power system. A VPP is able to generate the same amount of power of a large standard central power plant; however, it achieves this by aggregating and managing an entire network of DERs. Units in a VPP can be scattered across hundreds of private, commercial, and industrial locations, concentrated in a single area. This rich aggregation of micro energy assets is combined into a single entity, operating in the same manner of a conventional power plant, by means of a centralized control system. The VPP leverages the bidirectional flow of information of its smart grid: it receives power measurements, capacity, and availability information from DERs [2] and uses them to efficiently orchestrate utilities.

The VPP allows DERs to participate in the energy market and to provide grid ancillary services, such as power reserve or frequency regulation, allowing to enhance the power system's stability. It enhances both flexibility and stability in the power grid by providing services to better match supply/demand and allowing traditional utilities to plan and optimize production efficiency. Compared to a conventional energy management system, a VPP encourages a more dynamic and diverse energy market and it facilitates the use of cleaner energy sources. It increases economic returns for entities in the grid; this encourages more renewable installations, leading to a further push towards a sustainable energy supply. The VPP con operate so as to optimize energy flows over time, leading to economic savings both for consumers and for producers. Additionally, the structure itself of a VPP decreases energy loss in transportation because generation and consumption are localized in a specific area.

1.3 Energy Management System

The core of the intelligent VPP is the *energy management system* (*EMS*). The EMS manages loads, storages, and generators. It coordinates power flows among all entities and can perform forecasting and optimization for the entire grid.

This system leverages state-of-the-art techniques in information technology to identify optimum power dispatch, for example real time large data transfer, advanced forecast with smart algorithms and optimization strategies. Data science and machine learning algorithms are used to generate accurate predictions for power generation, load demand and electricity pricing forecast.

The EMS can balance provision fluctuations by turning up or down the energy supply to suit both the energy demand and the production. As energy demands constantly changes over time during the day, utilities must turn power on and off depending on the amount of energy needed at a specific moment.

VPP addresses uncertainty leveraging on the EMS. Power plants based on RESs introduce a new modus operandi named "*feed it and forget it*" [3] that adds complexity to the VPP operation. RESs have a stochastic and uncontrollable behavior; they produce and inject power into the electric network not following the demand, but according to external variables such as the time in the day, period in the year and weather conditions. This behavior makes it difficult to integrate the power generated by RES-based plants. Conventional power plants or large storage systems are used to balance both the demand of loads and the variable generation introduced by RESs. The smart grid provides the data and automation that enable RESs to put energy into the grid and optimize its use; the EMS plays a key role by managing all the entities interacting in this network.

The energy management system is also responsible for optimization in a VPP. It can operate by minimizing generation costs or maximizing profits. The cost of energy depends on availability and it fluctuates during the day; electricity is more expensive to provide at peak times because secondary - often less efficient - power plants must be operative to meet larger demands. Optimization algorithms in a VPP leverage power forecasts and compute the optimum projected power dispatch for all its energy assets. This enables units to operate optimally for maximum return. Based on the current energy price and the status of DERs, the EMS decides how much energy should be produced, which generator should be used to produce the required energy and whether the surplus energy should be stored or sold to the energy market external to the VPP. Sophisticated smart grids enable utilities, in cooperation with customers, to manage and moderate electricity usage especially during peak demand times, resulting in reduced costs for utilities and costumers. Moreover, by encouraging to defer electricity usage away from peak hours, electricity production is more distributed throughout the day, reducing costs and inefficient fluctuations. The EMS also performs forecasting on the smart grid systems, to predict and manage energy usage under different conditions and over time, leading to lower production cost.

The EMS plays an important role in assuring not only optimization but also stability and reliability of the grid. It manages electricity consumption in real time and receives continuous feedback information from DERs themselves. This greater insight allows the use of techniques to reduce, predict and overcome outages. Forecasting is used to predict energy fluctuations due to disruptions in the VPP caused by utility failures or weather conditions; the system can automatically identify problems in rerouting and restore power delivery.

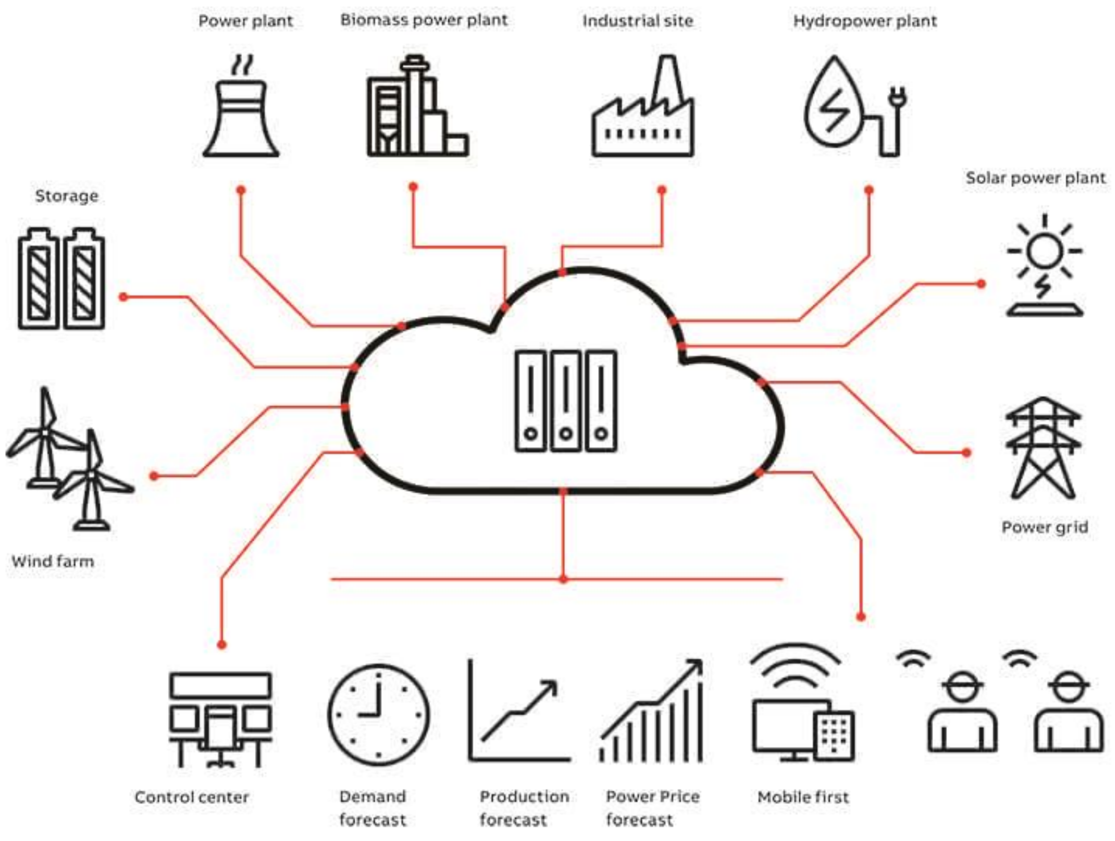


Figure 1*:* Virtual power plant schema. From ABB[1].

[1] *https://new.abb.com/power-generation/service/advanced-services*

Chapter 2
Optimization Under Uncertainty

The increasing amount and complexity of Distributed Energy Resources connected to the smart grid has brought some challenges to the management of the power system network. New optimization models are required to guide and control distributed units in the grid. In this context, virtual power plants play an important role by ensuring that the power produced, stored, and consumed by DERs is efficiently managed.

The energy management system of the VPP orchestrates entities and power flows with a certain objective, e.g. aiming at minimizing costs. In this process, the EMS considers several uncertainty factors that come into play in the smart grid, such as power generation from renewable resources. Uncertainty must be addressed so as not to compromise the reliability of the system. As a consequence, the optimization process performed by the EMS to decide power flows is a problem of making decisions under uncertainty, i.e. *stochastic optimization*.

This chapter provides information on methods for optimization under uncertainty and on how a VPP system can be modeled in optimization problems. These techniques represent the low-level optimizer inside the system proposed in this work.

2.1 Robust Optimization in VPPs

In [4] an optimization model to be employed in the EMS is presented. The proposed approach aims at minimizing operational costs by deciding the optimal planning of power flows for each point in time. It integrates into the model the necessary uncertainty elements.

The optimization approach is composed of two steps. The first is an offline day-ahead phase (a robust step) that computes optimized demand shifts to minimize the expected daily operating costs of the VPP. It uses a robust approach based on scenarios for modeling uncertainties present in the system, e.g. stemming from RESs such as wind or solar sources, and produces an estimated cost.

The second step is an online greedy optimization algorithm (*greedy heuristic*). It receives as inputs the optimized load shifts and it manages power flows in each timestamp in the VPP based on the real situation, with the aim of reaching the optimal real cost. This approach uses the optimized shifts produced by the first step to minimize, for each timestamp, the real operational cost, while allowing to fully cover the optimally shifted energy demand and avoiding the loss of energy actually produced by RES generators. It computes the real optimal value for the power flow variables based on the actual realization of uncertain quantities, assuming that the shifts have been planned by leveraging the first offline step; each timestamp is optimized one at time.

The first robust step produces good optimized shift that do not significantly deviate, in terms of cost, from the model with no uncertainty. On the other hand, according to the results in [4], the greedy step causes a significant loss in the quality of results.

2.2 Anticipatory Optimization Algorithms

A two-step optimization algorithm allows online decision-making in a VPP. The basic online step leverages a greedy heuristic that causes degradation in the quality of solutions, as mentioned in section 2.1. A way to improve the performance consists in replacing the greedy heuristic with a sampling-based stochastic anticipatory algorithm [5]. These algorithms were first developed for offline optimization, but they can be leveraged in online situations.

Offline applications are the usual focus for methodologies proposed in literature for stochastic optimization [6]. These methods usually base their optimization process on building a statistical model of future uncertainty, leveraging a sampling process that yields a set of scenarios. The Sample Average Approximation method [7] [8] solves stochastic optimization by adopting a simulation approach based on Monte Carlo simulation. It approximates the expected objective function of the stochastic problem with a function estimate on random samples; then, it solves with deterministic optimization techniques the resulting sample average approximation problems, in order to obtain candidate solutions for the original stochastic problem. This approach finds robust solutions by relying on one copy of the decision variables for each scenario and linking them via non-anticipativity constraints. It converges under reasonable assumptions and outperform greedy approaches.

Recent computational improvements in resources and techniques allowed the application of similar approaches to online optimization, leading to *stochastic online anticipatory algorithms*. Optimization problems under uncertainty usually benefit from an online approach; uncertainty progressively resolves in the online phase and decisions are made reacting and adapting to actual external events, allowing the discovery of robust high-quality solutions. An algorithm is called *anticipatory* if at some point it anticipates the future, namely, it makes use of information on the future to make decisions. While the algorithm at each time stamp can not fully know the future, i.e. the situation in following timestamps, it makes decisions based on inputs and possible future outcomes; future outcomes are estimated relying on possible scenarios delineated by past observations and current inputs.

Online anticipatory algorithms are effective [9] [10] but often computationally expensive, making them problematic as online decisions must be taken in short time frames. They usually rely on sampling to generate scenarios that estimate possible developments for a fixed number of future steps, called *look-ahead horizon*. Larger sample size leads to higher accuracy but also bigger problems to solve. This represents a problem because many use cases prescribe to make online decisions under strict time constraints. As a consequence, methods must be adopted to improve the efficiency of these methodologies; for instance, a conditional sampler can be exploited to generate scenarios taking into account past observations [11].

In many situations a significant amount of information is available before the online execution, in an *offline phase* where time constraints are relaxed. For example, an EMS might have access to energy production and consumption forecast for the smart grid. This *offline information* can be exploited for characterizing uncertain elements, for sampling likely outcomes, called *scenarios*, and for supporting online optimization strategies.

2.3 Hybrid Offline/Online Anticipatory Algorithms

Optimization under uncertainty can combine an online and an offline phase in order to achieve good solution quality with minimum online cost. A simple approach to tackle such problems is to deal with the offline and online phase separately, respectively via a sampling-based method and a heuristic. However, [12] [13] show that substantial improvements can be obtained by treating these two phases in an integrated fashion.

In particular, [13] proposes three methods that leverage an offline preparation phase to reduce the online computational cost of a sampling-based anticipatory algorithm, while maintaining the quality in the solution. The methods build on an online sampling-based anticipatory algorithm but shift part of the computation to an offline stage. The proposed hybrid offline/online approaches combine:

1. A technique to identify the probability of future outcomes based on past observations.
2. An expensive offline computation of a *contingency table*; it contains pre-computed solutions to guide online choices.

3. An efficient *solution-fixing heuristic* that adapts the pre-computed solutions to run-time conditions; it represents the core of the online computation.

These hybrid offline/online approaches are highly generic, i.e. they can be applied to any generic stochastic anticipatory algorithm.

The system devised in our work builds on one of the approaches proposed in [13] named $CONTINGENCY$, that leverages a contingency table containing *robust* solutions.

2.3.1 Modeling Online Stochastic Optimization

Online stochastic optimization is modeled as an n-stage problem, where at each stage some uncertainty gets resolved and some decisions are made. Each stage k (starting from $k = 1$ to n) is associated to a state variable s_k that summarizes the effect of past observed uncertainties and decisions, and a decision variable x_k that represents the decision taken. Uncertainty is modeled through a set of random variables ξ_i and it is assumed to be exogenous, i.e. it is only influenced by external factors and not by decisions. At each stage some random variables are observed; a $peek$ function determines which variables are observed depending on the state at that stage, and it returns a set O of indexes of the observed variables:

$$O = peek(s_k). \tag{1}$$

The set of unobserved variables is denoted as $\bar{O}$. Hence, among the random uncertainty variables, ξ_O denotes observed and $\xi_{\bar{O}}$ denotes unobserved ones.

2.3.1.1 Sampling-based Anticipatory Algorithm

The hybrid method starts from a given online sampling-based anticipatory algorithm, with the aim of reducing its online computational cost. It can be applied to a generic algorithm because it takes the algorithm itself as input.

An *anticipatory algorithm A* is *sampling-based* when it estimates future outcomes by leveraging scenarios. A *scenario* ω is a possible situation and it specifies a value ξ_i^ω for each random variable. The A algorithm determines the decisions x_k at stage k based on a set of scenarios Ω, the system state s_k and values for the observed uncertainty ξ_O:

$$x_k = A(s_k, \xi_O, \{\xi_\omega\}_{\omega \in \Omega}). \tag{2}$$

The state transition function $next$ determines the next state after the decision is established, given the system state s_k, the decision taken x_k and observed uncertainty ξ_O:

$$s_{k+1} = next(s_k, x_k, \xi_O). \tag{3}$$

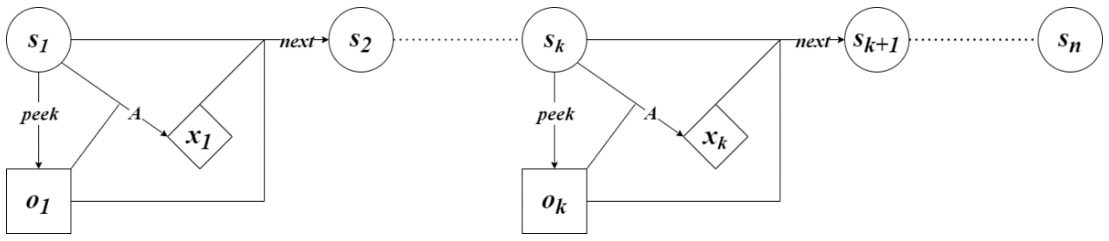


Figure 2: Online stochastic optimization is modeled as an n-stage problem. All the functions involved in the model are represented: *peek, next,* and *A*. At stage k: s_k is the system state, x_k the decision taken, and o_k are the observed variables - the related observed uncertainty is ξ_O -.

2.3.1.2 Base Behavior

The anticipatory algorithm A is an important component, but not the only one, of the online behavior of a system that includes stochastic optimization in an n-stage problem.

Given the initial state s_1, indices of observed variables O initially empty, a set of scenarios Ω and the random variables ξ representing uncertainty in the system, each online step k involves different phases. First, uncertainty is observed: a set of values, sampled from ξ_k based on $peek$ and on the state s_k, go from unobserved ($\xi_{\bar{O}}$) to observed (ξ_O). Then, the anticipatory algorithm A outlines the decision x_k and finally the next state s_k is determined via the $next$ function. The following pseudocode outlines this behavior and is referred hereinafter as $ANTICIPATE$:

Algorithm 1 $ANTICIPATE(s_1, \xi)$

```
Requires:
    $\Omega$ : set of sampled scenarios
    $O = \emptyset$ : indices of observed variables

for $k = 1, \dots, n$ do
    $O \leftarrow O \cup peek(s_k)$
    $x_k \leftarrow A(s_k, \xi_O, \{\xi_\omega\}_{\omega \in \Omega})$
    $s_{k+1} \leftarrow next(s_k, x_k, \xi_O)$
return $s, x$
```

This behavior represents the hefty anticipatory algorithm originally adopted in the online step. However, its application is not necessarily limited to the online phase. In fact, the hybrid method explained in the following sections is based on the idea of shifting the expensive computation of $ANTICIPATE$ to an offline stage.

2.3.2 Offline Information and Scenario Sampling

Offline information I is defined as a collection of observed uncertain values and it can be exploited to support online optimization. In many use cases it is possible to have access, during the offline phase, to information such as historical data, data from simulations, predictions and forecasts. Following its definition, offline information I is a collection of (observed) scenarios ω. We assume that I is representative of the actual probability distribution of the random variables.

The set of scenarios Ω involved in the sampling-based anticipatory algorithm must be as representative as possible in order to maximize its effectiveness. Offline information I can be leveraged in order to define such a representative Ω set: Ω can be obtained via random uniform sampling from I.

In a stochastic optimization problem, uncertainty progressively resolves itself as random variables are observed at each stage. If variables ξ_i are not statistically independent, a set of scenarios Ω that was relevant at the beginning might lose its relevance when uncertainty is resolved. For instance, in a VPP a set of scenarios Ω involving power generation by wind plants is not relevant in a day with no wind detected.

Namely, we want a *conditional sampler* that generates scenarios consistent with past observations ξ_O, allowing us to sample at stage k the unobserved variables $\xi_{\bar{O}}$ according to the conditional distribution $P(\xi_{\bar{O}}|\xi_O)$. If scenarios are sampled from the offline information I, this effect is created by sampling based on the *conditional probability* of scenarios ω in I with respect to past observations; following the *fundamental rule* for probability calculus, this is computed as:

$$P(\xi_{\bar{O}}^{\omega}|\xi_O) = \frac{P(\ \xi_{\bar{O}}^{\omega}\ \xi_O\ )}{P(\ \xi_O\ )}, \omega \in I \tag{4}$$

Here, $P(\ \xi_{\bar{O}}^{\omega}\ \xi_O\ )$ is the *joint probability,* i.e. probability for observed and unobserved values to occur together, and $P(\ \xi_O\ )$ is the *marginal probability* for observed values, i.e. probability that these values are observed. Estimation of the joint probability can be obtained using a density estimation method, e.g. Gaussian Mixture Models [14] or Kernel Density Estimation [15]. Offline information can be exploited to train any of these methods and obtain an estimator $\tilde{P}(\ \xi_{\bar{O}}^{\omega}\ \xi_O)$, in short $\tilde{P}(\ \xi)$, for the joint distribution of random variables. On the other hand, the marginal probability can be computed from the estimator $\tilde{P}(\ \xi)$ through *marginalization,* i.e. aggregating the contribution of all unobserved variables $\xi_{\bar{O}}$:

$$\tilde{P}(\xi_O) = \sum_{\omega' \in I} \tilde{P}\left(\ \xi_{\bar{O}}^{\omega}\ \xi_{\bar{O}}^{\omega'}\right) \tag{5}$$

Hence, an estimator $\tilde{P}\left(\xi_{\bar{0}}^{\omega} \middle| \xi_{O}\right)$ for the conditional probability is:

$$\tilde{P}\left(\xi_{\bar{0}}^{\omega} \middle| \xi_{O}\right) = \frac{\tilde{P}\left(\ \xi_{\bar{0}}^{\omega}\ \xi_{O}\ \right)}{\sum_{\omega' \in I} \tilde{P}\left(\ \xi_{\bar{0}}^{\omega}\ \xi_{\bar{0}}^{\omega'}\right)}\ , \omega \in I \tag{6}$$

and it is proportional to the true probability value: $P\left(\xi_{\bar{0}}^{\omega} \middle| \xi_{O}\right) \propto \tilde{P}\left(\xi_{\bar{0}}^{\omega} \middle| \xi_{O}\right)$.

If scenarios are drawn from the offline information I following this probability rule, their distribution takes into account the effect of past observations, namely, observed variables.

2.3.3 Contingency Table

In the offline phase there are not strict time constraint or resource limits, e.g. parallelization can be exploited. Therefore, it is possible to reduce the computational cost of the online algorithm at the expense of adding a costly offline step. The offline information I can be exploited, if significant time is available in the offline phase, to perform an offline simulation of online situations, aimed at preparing for all possible developments. Each scenario ω in I is considered as if it was a real sequence of online observations; ω is fed to an anticipatory algorithm, for example the expensive online algorithm typically adopted in the online phase introduced in Section 2.3.1.2. This process produces a set of robust solutions, in form of a *contingency table*, that can be used as input data to guide a lightweight online method. The latter method is the only computation actually performed during the online phase. This approach results in a very expensive offline computation that allows significantly lighter online steps.

The offline process is referred to as $BUILDTABLE$. It takes as input the anticipatory algorithm AA, analogous to the anticipatory algorithm $ANTICIPATE$ and with the same input parameters, together with the initial state of the system s_1. For each scenario $\omega \in I$, AA is applied obtaining the sequence of states s^{ω} visited by the system and the sequence of decisions x^{ω} outlined by the algorithm. The *contingency table* T is the data structure resulting from this process: a pool of *traces*, namely, scenarios paired with information on state sequences and decisions.

Algorithm 2 *BUILDTABLE*(s_1, AA)

```
Requires:
    $I$ : offline information
    $T = \emptyset$ : contingency table

for $\omega \in I$ do
    $s^\omega, x^\omega \leftarrow AA(s_1, \xi^\omega)$
    $T \leftarrow T \cup (\xi^\omega, s^\omega, x^\omega)$
return $T = \xi^\omega, s^\omega, x^\omega{}_{\omega \in I}$
```

2.3.4 Fixing Heuristic

The augmented information contained in the contingency table is used online to guide the efficient *fixing heuristic*, whose purpose is to adapt pre-computed solutions to real online conditions. The fixing heuristic solves a light optimization problem, with the aim of selecting decisions that have the largest change of being optimal, based on the actual state and observations. The objective function is:

$$argmax\,\{P^*(x_k|s_k\xi_O): x_k \in X_k\} \tag{7}$$

where P^* represents the probability for the decision x_k to be optimal, conditioned by the state s_k and the observed uncertainty ξ_O, and X_k is the feasible decision space.

An estimation of P^* in the objective of the fixing heuristic can be obtained by leveraging the contingency table T. In short, the heuristic is translated, for discrete or numeric problems respectively, into the problem of minimizing the weighted Hamming or Euclidian distances with respect to traces in T. Complete proof of the process to obtain estimators for P^* is reported in [13].

We report here, for sake of completeness, how the objective function is materialized for the two categories of problems. We denote with a compact notation

$P(\omega)$ the probability that the same state as the trace ω is reached, and then everything goes according to the plan; it can be approximated as:

$$P(\omega) \propto \tilde{P}(\, s^{\omega}_{sk+1} \mid s_k)\tilde{P}\big(\, \xi^{\omega}_{\bar{0}} \mid \xi_O\big), \omega \in T \tag{8}$$

where $\tilde{P}\big(\, \xi^{\omega}_{\bar{0}} \mid \xi_O\big)$ is the estimator detailed in Eq. (5) and $\tilde{P}(\, s^{\omega}_{sk+1} \mid s_k)$ is a similar estimator for states that can be obtained with an analogous process.

- *Discrete problems.*

  The objective function for the fixing heuristic becomes:

$$argmin \left\{ -\sum_{j=1}^{m} \sum_{v \,\in\, D_j} log\, p_{jv} \,[\![x_{kj} = v]\!] : x_k \in X_k \right\} \tag{9}$$

  where $[\![*]\!]$ denotes the truth value of the predicate *, D_j is the domain of x_{kj} and v is one possible value for it. The probability p_{jv} for the j-th value and v is estimated as:

$$p_{jv} \;=\; \frac{\sum_{\omega \,\in\, T,\, x^{\omega}_{kj} = v} P(\omega)}{\sum_{\omega \,\in\, T} P(\omega)} \tag{10}$$

- *Numeric problems.*

  The objective function for the fixing heuristic becomes:

$$argmin \left\{ \sum_{j=1}^{m} \sum_{\omega \,\in\, T} p_{\omega} \, \frac{1}{2\sigma_j} \big(x_{kj} - x^{\omega}_{kj} \big)^2 : x_k \in X_k \right\} \tag{11}$$

  Where the probability p_{ω} is estimated as:

$$p_{\omega} \;=\; \frac{P(\omega)}{\sum_{\omega' \in T} P(\omega')} \tag{12}$$

The fixing heuristic is the core of the highly efficient online step. Intuitively, the behavior of the online phase with the heuristic follows a similar approach to the one in $ANTICIPATE$. First some uncertainty is observed, then a decision is outlined, and finally the next state is computed; the difference is the peculiar logic adopted to take decisions. Its pseudocode is reported below:

Algorithm 3 *FIXING*(s_1, ξ, T)

```
    Requires:
        objective : objective function for the
                    heuristic, as in Eq.
                    (7), (9) or (11)
        $O = \emptyset$ : indices of observed variables

    for $k = 1, \ldots, n$ do
        $O \leftarrow O \cup peek(s_k)$
        $\Omega \leftarrow top\ elements\ \omega \in T\ by\ descending$
                        $probability\ P(\omega)\ according\ to\ Eq. (8)$
        $p_{jv}\ or\ p_{\omega} \leftarrow compute\ Eq. (10)\ or\ (12), based\ on\ \Omega$
        $x_k \leftarrow solve\ objective\ in\ Eq. (9)\ or\ (11)$
        $s_{k+1} \leftarrow next(s_k, x_k, \xi_O)$
    return $s, x$
```

2.3.5 Hybrid Offline/Online Method

The low-level optimizer in our system is a hybrid offline/online technique for optimization under uncertainty. We adopt a methodology proposed in [13] that combines the methods introduced in sections 2.3.2 to 2.3.4.

The hybrid offline/online algorithm adopts the contingency table and the fixing heuristic. The main idea is to leverage the offline step to compute *robust solutions* for all scenarios ω in the offline information I, obtaining the contingency table T. Then, in the online step these augmented data are used as a guidance for the efficient solution-fixing heuristic $FIXING$, that takes into consideration the real online situation. Robust solutions are obtained using $BUILDTABLE$, detailed in section 2.3.3, where the (expensive) $ANTICIPATE$ is adopted as the anticipatory algorithm AA. In other words, the anticipatory algorithm $ANTICIPATE$ is used offline.

In this setting, the aim of the fast fixing heuristic is to match the quality of robust solutions obtained via the expensive anticipatory algorithm $ANTICIPATE$. Intuitively, the anticipatory algorithm usually employed online is moved offline; the online step leverages a much lighter optimization problem whose aim is to match the quality of the offline solution. The cost to pay for the significant reduction of online cost is the introduction of a heavy offline step.

Pseudocode for the hybrid offline/online method is reported below.

Algorithm 4 *CONTINGENCY*(s_1, ξ)

```
Requires:
    $O = \emptyset$ : indices of observed variables

$\tilde{P}(\xi)$ ← train estimator for the joint
      distribution of random variables on
      offline information $I$
$T$ ← BUILDTABLE($s_1$, ANTICIPATE)
$\tilde{P}(s_k\, s_{k+1})$ ← train estimator for the joint
      distribution of states on $T$, for all
      steps $k$
$s$, $x$ = FIXING($s_1$, $\xi$, $T$)
return $s, x$
```

2.3.6 VPP Model

A Virtual Problem Plant aggregates and manages power generation, storage and load units. The energy management system orchestrates them: it decides power flows with the aim of satisfying the power demand of loads, respect regulations and physical limits, and minimize the operating costs [16] [17]. The uncertainty factors that come into play in this system are the generation from Renewable Energy Sources and the demand by load units.

The EMS optimization problem in a VPP can be translated in terms of an optimization problem under uncertainty by specifying all variables and functions that come into play. In particular:

- The sampling-based anticipatory algorithm for making decisions A.
- The decision, state, and random variables; respectively x, s and ξ.
- The $peek$ and $next$ functions.
- The feasible space for decisions X_k.
- A cost metric that allows one to evaluate the quality of solutions.
- A technique for obtaining the probability estimator $\tilde{P}$.

The low-level optimizer of the system proposed in this work leverages such an optimization problem to model the controlled VPP. The model is introduced in [13]. Complete information about it can be found in the related repository[2].

The sampling-based anticipatory algorithm A adopted as the basic algorithm is a Mathematical Programming model based on the Sample Average Approximation.

The decision at stage k is represented by a decision vector x_k. Its components specify the power flow x_{kj} through each node j in the system - generation, storage or load units; for example, x_{kS} indicates the power flow for the storage unit. The state component s_{kS} refers to the power level stored in this unit, while s_{kD} gives information about its flow direction. The random variable ξ_k has components for each uncertainty factor: ξ_{kR} corresponds to the RES generation and ξ_{kL} to the load.

[2] *https://github.com/alleDe/OffOn*

The $peek$ function decides which random variable to observe at each stage; it returns R and L for stage k:(k, R) and (k, L). The $next$ function incorporates the logic for the state change. The storage charge level at stage $k+1$ is proportional to the charge level and the storage power flow in the previous stage k, with a dependency on the charging efficiency of the storage unit η. The flow direction at $k+1$ depends on the direction of the power flow at stage k. Formally:

$$s_{k+1,S} \; = \; s_k \; + \; \eta x_{k,S} \tag{13}$$

$$s_{k+1,D} \; = \; 0 \; if \; x_{k,S} \geq 0, 1 \; otherwise \tag{14}$$

The set X_k, representing the feasible decision space, is defined by a separate problem with its constraints and variables. Its objective and constraints enforce power balance in the system and physical limits for units and power flows. The corresponding mathematical program is:

$$\xi_{kL} \; = \; \sum_{j=1}^{m} x_{ji} \; + \; \xi_{kR} \tag{15}$$

$$l_j \; \leq \; x_{kj} \; \leq \; u_j \quad , j = 1, \dots \, m \tag{16}$$

$$0 \; \leq \; s_k + \; \eta x_{k,S} \; \; \leq \; \Gamma \tag{17}$$

$$\xi_{kL} \; = \; \sum_{j=1}^{m} x_{ji} \; + \; \xi_{kR} \tag{18}$$

$$x_k \; \in \; \mathbb{R}^m \tag{19}$$

Each power flow is associated to a cost c_{kj} at stage k. The storage unit is associated to a cost as well, related to its wearing off; this cost occurs when the flow direction in the storage system changes and it is proportional to a cost value α. Hence, the total operational cost incurred at stage k is modeled as:

$$\sum_{j=1}^{m} c_{kj} \, x_{kj} + \alpha \left| s_{k,D} - s_{k+1,D} \right| \tag{20}$$

It is worth to note that the cost term related to storage wear-off implies that the anticipatory algorithm A must solve an *NP-hard* problem, whereas the fixing heuristic does not.

Kernel Density Estimation [15] with Gaussian Kernels is the technique adopted for computing approximations of the probability distributions. It is used for obtaining the estimator for the joint distribution of the random variables $\tilde{P}(\xi)$ and its derivates, and in a similar computation for computing the estimator $\tilde{P}(s_k s_{k+1})$ and its derivates.

2.3.7 Execution and Data Generation

The execution of the hybrid offline/online approach based on contingency table generates data that are used as to build the high-level optimizer. This section provides detail on how these data were obtained.

2.3.7.1 Experimental Setup

The experimental setup to generate the data is similar to the one introduced in [13].

The hybrid offline/online approach is applied on real *instances* for the virtual power plant system. An instance is a specific realization of uncertainty in the system, i.e. a sequence of realizations for the stages. Uncertainty realization is obtained by sampling values for the random variables associated to RES generation (*PV*) and loads (*Load*). Sampling is performed so as to ensure statistical independence between variables, similarly to a realistic situation. The result of this process is the offline information *I* and the sequence of observations, namely, a sequence of values for *PV* and *Load* for all stages.

The problem is modeled using real physical bounds for power generation, realistic power flow limits, initial battery state, efficiency, according to [17] [18]. The time frame for the whole optimization problem is a full day (24 hours), and two subsequent stages are 15 minutes apart; the electricity price is also assumed to change every 15 minutes. Therefore, the *horizon* for the optimization problem involves 96 stages – 4 stages x 24 hours.

The baseline method adopted for comparing the hybrid offline/online approaches is a myopic (greedy) heuristic. In this setting it is represented by the anticipatory algorithm $ANTICIPATE$ run with an empty set of scenarios, formally $\Omega = \emptyset$.

The instances used in the experiments that generate our dataset are 100. Each instance is fed as input to the optimization approaches 100 times, varying the number of traces in the contingency table T from 1 to 100. For each run the following data are recorded:

- Sequence of realizations for the variables PV and $Load$ in all stages, i.e. information on the instance.
- $nTraces = |T|$, number of traces in the contingency table T used in that run.
- Cost of the solution found by the approach. Lower cost indicates a better solution quality.
- Time required by the approach for online computation.
- Average memory used during the online computation.
- Maximum memory used during the online computation.
- Average CPU amount used during the online computation.
- Maximum CPU amount used during the online computation.

These experiments yield a set of 100 x 100 = 10000 entries. It is the dataset used in the following sections.

2.3.7.2 Results

Results reported in [13] show that the hybrid offline-online approach substantially reduces the computational time of the online phase, at the expense of a hefty offline step. At the same time it achieves high solution quality, comparable with the anticipatory algorithm.

According to the results, there is a noticeable tradeoff between online computational time and solution quality in the optimization methods. Experiments compare the baseline (greedy heuristic), $CONTINGENCY$ and the original anticipatory algorithm ($ANTICIPATE$) when adopted as optimization approach in the online phase. The greedy heuristic is outperformed by all optimization methods by a significant margin. $CONTINGENCY$ leads to a significant reduction in online time expense compared to $ANTICIPATE$ and it yields solutions whose cost is worse but remarkably close to the original $ANTICIPATE$ algorithm.

Increasing the number of guiding traces leads to a decrease in the solution value (i.e. worse quality) in $CONTINGENCY$. The online cost has a significant increase for $ANTICIPATE$ and when the number of traces increases, while it slightly rises for $CONTINGENCY$. Namely, the gap in time performance gets larger with the number of scenarios.

There is room for improving the applicability and efficiency of hybrid offline/online methods. A fundamental direction to explore is how to determine the number of guiding traces for the fixing heuristic. The number of traces is not fixed, and the optimal value can depend on the actual condition of the problem to be optimized. Furthermore, the study reported in [13] underlines the cost/quality tradeoff between online computational time and solution quality. Namely, increasing the number of traces leads to better solution quality but degrades the online time required by the methods.

This motivates the system designed hereinafter. We design and construct a system that builds on the hybrid offline/online approach based on fixing heuristic and contingency table, i.e. $CONTINGENCY$. The proposed system is aimed at automatically suggesting the optimal algorithm configuration (i.e. number of traces) based on the instance of the problem to solve, taking into account constraints such as desired solution quality and time and memory availability. The system has a more general purpose than a suggestion system: not only it suggests the configuration of the hybrid optimizer, but it can also provide forecasts about its performance or required online time and resources. Moreover, the system is designed to be highly flexible and thus it can work in the opposite direction. For example, it can provide an estimation of required time and resources for an instance when the optimization method must reach a desired solution quality.

Chapter 3
Machine Learning Models

Variables involved in the VPP optimization problem interact through complex non-linear relationships. For this particular reason the relationships among variables are captured via machine learning models. They allow to express highly complex relationship and they provide large flexibility in modeling.

The proposed system focuses specifically on decision trees an artificial neural networks. These models are suitable for modeling the runtime behavior of optimization algorithms [19]. Moreover, they perform good modeling on our dataset, as proved in next sections, and they are supported by EML.

3.1 Machine Learning

Machine learning (ML) is a subfield of artificial intelligence (AI) that is heavily grounded on mathematics and statistics. In recent years, with the increase of computational resources and amounts of data produced and consumed, ML has experienced a growing importance in computer science and sciences in general, becoming ubiquitous in many fields. At a high level, ML is aimed at detecting patterns in data and using them to make predictions and decisions. It is useful for automating analysis and tasks that humans usually perform, often outperforming usual human-based approaches. Compared to classic statistics, ML is generally focused on large datasets, on making predictions and on providing highly flexible models.

The construction of ML models typically involves several steps. They have been adopted for the development of our system:

1. Collection of background information on the application and the task.
2. Data collection. Data are the core of ML models.

3. Data preprocessing: cleaning, preprocessing, transformation. A thorough preprocessing of data is a fundamental step to ensure the success of ML techniques.
4. ML algorithm/model selection, based on the specific application and data themselves.
5. ML model building and evaluation.
6. Use of results. Use either the ML model itself, or patterns and predictions obtained with it.

Data involved in ML problems are usually organized in a table, where:

- Each row is a data point from the set, usually called *example* or *sample*.
- Each column is called *feature* and it represents a measure that can be performed on samples.

Machine learning techniques are typically grouped in two major categories:

- *Supervised*. In a supervised model, the input for an example is a set of features and the output is the desired class label or value. The general supervised learning problem takes as input examples with features and corresponding labels/values; it produces as result the ML model that predicts the label/value for new unseen examples. Hence, the goal of supervised ML is to use data to find a model that outputs the right label or value based on input features. The operation performed by these methods is not searching but learning, as a model must be able to operate on unseen data.

  If the objective of the prediction is:

  - a categorical label, the task is called *classification*.
  - a numeric value, the task is called *regression*.

  This framework is highly generic: it can be applied any problem that involves any input/output mapping.
- *Unsupervised*. In the general unsupervised learning problem, no class labels or values are given for the input data. The goal of unsupervised ML is to find important patterns in data or to associate data points to meaningful labels.

When an unsupervised model is constructed only features are available, with no explicit target labels/values, and we perform computations on them that are useful for the specific task. There are several things that we might want to do in an unsupervised ML task: clustering, outlier detection, similarity search, visualization, ranking and so on.

The way our problem is structured makes it a typical use case for supervised machine learning.

3.1.1 Building a Supervised Model

3.1.1.1 Training

The construction of a ML model is called *training* and it yields a custom model that fits the data used to build it. The input of this phase is called *training set*; in supervised ML, it is the dataset of features and target values. The output is a *trained* model. At *inference* time, the model takes only features as inputs.

Given a specific category of ML models, training can be interpreted as searching among all possible model's values and parameters. It is the task of finding the model with the correct values and parameters that better fits the training set. The quality of this fit can be assessed in several ways. Accuracy is usually adopted in classification tasks; on the other hand, some formulation of error or distance is the typical choice for regression, e.g. mean squared error. In other words, training is equivalent to minimizing the training error or maximizing the accuracy.

3.1.1.2 Testing

A ML model must be able to *generalize*, i.e. to make predictions on unseen data rather than training data. This is the fundamental difference between the ability to learn and to memorize. A model might generate perfect predictions on the training set, achieving a training error of 0. However, it does not necessarily have the same performance on new data. The phenomenon of obtaining a (significantly) larger error on new data compared to the one on the training set is called *overfitting*. In this situation the model is too specific to the exact training set, and it does not generalize well; it might have captured patterns in the training set that are noise rather than general characteristics. Overfitting is more likely when the model is complicated or the amount of training data is low. On the other hand, a model that is too simple does not capture significant patterns in the training data; in this situation, named *underfitting*, also the training error is significantly large. It is fundamental to test models on data not used in the training phase in order to detect these two opposite situations.

The construction of a supervised learning model involves two steps:

1. *Training phase*: build the model based on training data. The *training error* assesses the model's prediction on this set.
2. *Testing phase*: the model makes predictions on test data, i.e. data not seen in the training phase. Usually the test set is similar to the training set: samples characterized by features are associated to their target values. The *test error* evaluates the model's performance on this set, i.e. how far its predictions are to the real targets. The test error is a better estimate of the model's ability to generalize.

The goal of machine learning is to learn rather than memorize, i.e. perform well in new situations. Therefore, it is fundamental to perform both steps, and the real indicator of a model's quality is test error.

Test data can not influence the training phase in any way; this is sometimes referred to as the *golden rule of machine learning*. The test error measures the performance of the model on new data. If the test set contains samples used during training, the assessment is not accurate and the model overfits part of the test data.

Defining a ML model requires both training and test sets to be carefully constructed in order to allow a fair evaluation of performances. The construction of models in the experiments hereinafter follow these fundamentals machine learning practices.

3.1.1.3 Validating

Machine learning models are characterized by parameters that control how well the model fits the dataset; these are the parameters that during the training phase are adjusted to find the best model for the training set. ML models also have *hyper-parameters*, for example the tree depth in a decision tree. Hyper-parameters generally control the model's complexity. They are not learned in the training step: it is always possible to fit the training data better (i.e., lower training error) by making the model more complex, but this inevitably leads to overfitting. Hence, values for hyperparameters are fixed before training a model.

Values for hyper-parameters should be selected in order to achieve the lowest test error possible. However, using test data in the construction of a model – even for choosing hyperparameters – violates the golden rule of ML. To overcome this situation, part of the training data is usually kept separate and used as a surrogate test set at training time; in this setting, the training set is split in two sets:

- *Training set*: data actually used to train the model.
- *Validation set*: used to test the model. The error represents an approximation of the test error.

Hyper-parameter selection is performed with the guidance of the *validation error*. Several models with different candidate values for hyper-parameters are trained on the limited training set and tested on the validation set. The set of hyper-parameters achieving the lowest validation error is selected. Usually, after choosing hyper-parameters, the final model is trained on the entire training set in order to exploit the full dataset when fitting it.

Cross-validation is a technique aimed at improving the validation step. In k-fold cross-validation, the entire training set is split into k subsets of roughly the same size called *folds*. A model is trained on k-1 folds and validated on the remaining one. This process is repeated k times, where each iteration adopts a different fold as validation set. The final *cross-validation error* is the average validation error across all k iterations; it represents a more accurate approximation of the test error compared to standard validation. With this approach, every training sample contributes in validation and training without violating the golden rule of ML; at the same time, a large portion of the dataset (k-1 folds over k) is used for training models. The pitfall is a larger computational cost, as each cross-validation score involves the training of k models; this score is computed for each hyper-parameter set to evaluate, hence the hyper-parameter selection process becomes expensive. A larger number of folds yields a more accurate error estimation but a higher computational cost. In *leave-one-out cross-validation*, one training example is used for validating each model.

Machine learning models involved in our system adopt the cross-validation approach for selecting hyper-parameters. The process of constructing a ML model involves the separation of data into a training and a test set. The first is used for building the model, the latter is kept apart and used for testing at the very end. In order to select hyper-parameters, within the cross-validation process, the training set is split again into an actual training set and a validation set. Once hyper-parameters are selected, the model is trained on the entire training set. This final model is used at inference time to perform predictions.

3.2 Decision Trees

A *decision tree* (*DT*) is a supervised machine learning model consisting of a nested sequence of if-else decisions, called *splitting rules*, based on the features. A class label or a numerical value is returned at the end of each sequence. The first rule – at depth 0 - is called *root*, intermediate rules are called *nodes*, and the final parts are the *leaves* of the DT.

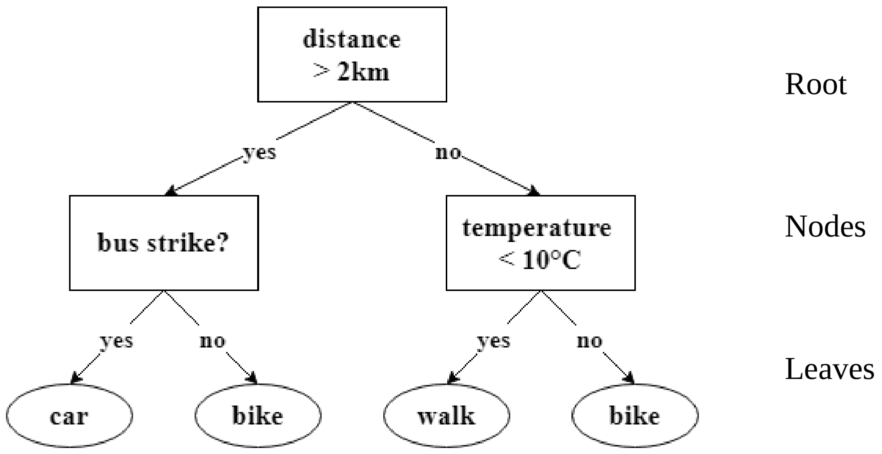


Figure 3*:* Decision tree with depth two. The target of the prediction is the means of transport to reach a destination – a categorical target. Features are the distance to the destination, the weather temperature on that day and the presence of a bus strike.

3.2.1 Decision Stump

The building block of a DT is a simple splitting rule based on thresholding one feature. A DT composed of only one splitting rule is also called *Decision Stump* (*DS*). In the training phase of a DS, the aim is to find the best rule to fit the training set – namely, a feature, a threshold, and leaf values. This is achieved by first defining a score that evaluates the quality of the model, then searching for the rule that yields the best score. Hence, training is reduced in searching among all possible rules the one with best score. Usual scores adopted during this search are accuracy for classification and mean squared error for regression.

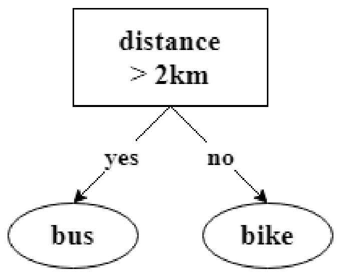


Figure 4*:* Decision Stump. Same classification problem of Figure 3: the target of the prediction is the means of transport to reach a destination.

3.2.2 Decision Tree

DTs are an extension of DSs that allow sequences of splits based on multiple features. While a DS has small expressive capability and limited accuracy, a DT is more general and can achieve large accuracy even in complex scenarios.

It is computationally infeasible to train a DT with the same approach used for DSs, namely, by exhaustively searching for the best DT among all possible sequences of rules. The most common DT learning algorithm reported in literature, called *Greedy Recursive Splitting* (*GRS*), addresses this issue. Starting from the full training set, GRS trains a DS on it. The original set is split by the DS's rule into subsets, one for each leaf; GRS fits an additional DS on each leaf's data, resulting in a depth-2 DT. This process is repeated for increasing depths until a *stopping criterion* is met. Several stopping criteria are reported in literature; for example, a leaf has few samples or it only has one label, no rule improves accuracy/error on the resulting sets or a user-defined maximum depth is reached. Each leaf in the resulting DT is associated to a label or a value based on the training samples that end up in that leaf.

Depth is a fundamental hyper-parameter to handle while fitting a DT. A DS is a DT with depth 1. The larger is the depth, the more complex is the model, hence the larger is the risk of overfitting.

At inference time a sample is fed to the DT. The values of its features determine a path inside the tree, i.e. a sequence of decisions leading to a leaf. The DT's prediction for the sample is the value associated to that leaf.

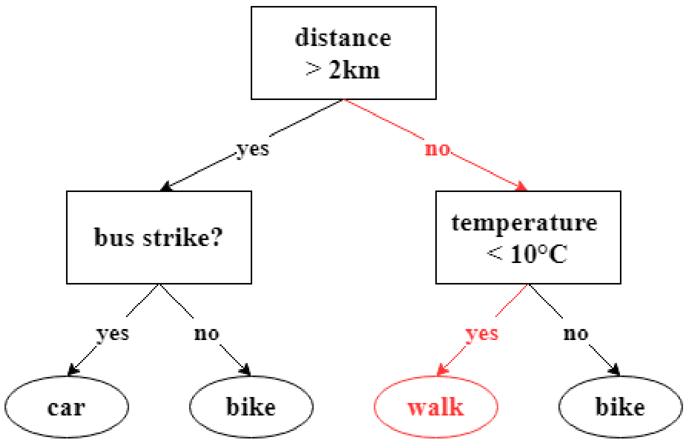


Figure 5: Inference in a Decision Tree. In red the sequence of decisions for a sample that is labeled as "walk".

Decision trees are some of the easiest ML models, yet very used in real-world applications. They are highly interpretable: even non-experts can examine a DT structure and understand what happens. They are easy to implement compared to other ML models, their learning is fast and inference is very fast. Moreover, they can handle both categorical and numerical data, and can elegantly deal with missing values in the training set. They do not require input data in specific formats hence there is no need for special data pre-processing. The major pitfall of these models is the difficulty of finding the optimal set of rules; GRS is often not accurate and might require very deep trees that easily overfit. As DTs are usually prone to overfitting, stopping criteria and mechanisms such as *pruning* must be adopted to allow good generalization capabilities. Ensemble versions of DTs are often adopted to reduce overfitting and are called random forests.

The remaining part of this section provides a technical explanation of classification and regression DTs. They are some of the ML models adopted in our experiments.

3.2.2.1 Classification

The score criterion commonly used to train rules (i.e. select splits) in a classification DT is *information gain*: the selected split is the one that decreases *entropy* of labels the most. Entropy measures the randomness of a set of data, namely, how many bits of information are encoded in the average sample. Low entropy indicates a very predictable set carrying small information whereas large entropy indicates randomness. If set has k classes and p_c is the probability of each class c, the entropy is:

$$entropy(set) = -\sum_{c=1}^{k} p_c log(p_c) \tag{21}$$

Assume y is the set of labels for the training samples, with cardinality n. A rule splits y into two subsets y_{yes} and y_{no}, with n_{yes} and n_{no} examples respectively. Information gain for the split is:

$$InfoGain = entropy(y) - \frac{n_{yes}}{n} entropy(y_{yes}) - \frac{n_{no}}{n} entropy(y_{no}) \tag{22}$$

Information gain is large if labels are “more predictable” (“less random”) in the next layer. Even if a split does not increase classification accuracy at one depth, the hope is that it makes classification easier at the next depth, as the resulting sets are less random.

In a classification DT each leaf is associated to a class label. Its value is usually the mode of labels for the training samples in that leaf, i.e. the most common training label.

3.2.2.2 Regression

Regression DTs follow a similar approach compared to classification DTs while dealing with numerical predictions. Several functions are reported in literature to measure the quality of a split. *Mean Squared Error* (*MSE*) and *Mean Absolute Error* (*MAE*) are two of the most common criteria, respectively defined as:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{23}$$

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \tag{24}$$

where the training set to split has n samples, each of them has true target value y_i and prediction after the split $\hat{y}_i$.

In a regression DTs each leaf is associated to a numerical value, for example the mean or median of target values for samples in the terminal node.

3.2.3 Random Forest

In general, *ensemble methods* are models that have models as input, in a framework sometimes referred to as *meta-learning*. The aim of ensemble methods is to obtain *meta-models* that achieve higher accuracy or lower error compared to the input ones. Some ensemble techniques, named *boosting-based methods*, start from models that easily underfit and are aimed at improving the training error. On the other hand, *averaging-based methods* build on models that overfit and their goal is to limit overfitting.

Random forests (*RFs*) are the ensemble version of decision trees. They average a set of deep randomized DTs with the goal of increasing performance and controlling overfitting. The idea is that a single DT makes error in predictions; however, aggregating the result of multiple different DTs yields better performance, as errors of a single DT are corrected by all the others. Randomization must be included in the construction of RF in order to obtain different trees with independent errors. Two ingredients are fundamental for providing randomization:

- *Bagging*: using *bootstrap samples* for ensemble learning. A bootstrap sample of the original training set is a set of the same size, obtained by sampling with replacement from it. Each DT in the ensemble is trained on a different bootstrap sample, hence it will have different splits.

- *Random Trees*: when training a DT, each split is decided within a random subset of the features instead of all possible features. Different trees will have different splits as the subset of feature to search at each split is random.

The meta-model predicts a single value that aggregates the outputs of all models, usually the majority-predicted label for classification and the average output value for regression. Although single DTs in a RF are affected by overfitting, if their errors are independent aggregating predictions hides the single weaknesses. This results in a better generalization capability, i.e. lower test error.

3.3 Neural Networks

Artificial neural networks (*ANNs or NNs*) are supervised learning models and are the core of *deep learning*. A trained NN model performs a non-linear transformation that represent a complex mapping between an input and an output vector. A NN is a composition of simple functions in multiple layers connected together via weights and non-linearities. More layers yield more complex mappings.

NNs are divided into two main categories, depending on their architecture:

- *feed-forward neural networks* if neurons are connected in an acyclic graph – signals only proceed forward in the network;
- *recurrent neural networks* that introduce cycles in neurons connections.

We focus our attention on the first as they are adopted in our experiments.

The increasing amount of available data and computational capabilities has contributed to the growth of NNs' popularity in recent years. NNs are currently the base of state-of-the-art systems in Computer Vision and Natural Language Understanding, yielding unprecedented performances on difficult tasks.

3.3.1 Neuron

The *neuron* is the basic unit of computation in a NN and it represents a function. It accepts some input signals in form of a vector x, it computes their weighted sum and it adds a bias b. Finally, it applies a (non-linear) *activation function* h, also referred to as *non-linearity*; the argument of the activation function is sometimes called *neuron activity*. Weight values are learned during the training process. Formally, assuming x is the n-dimensional input vector and w is the vector representing weights, the output is:

$$y = h(b + w^T x) = h\left(b + \sum_{j=1}^{n} w_j \, x_j \right) \tag{25}$$

$$a = b + w^T x \tag{26}$$

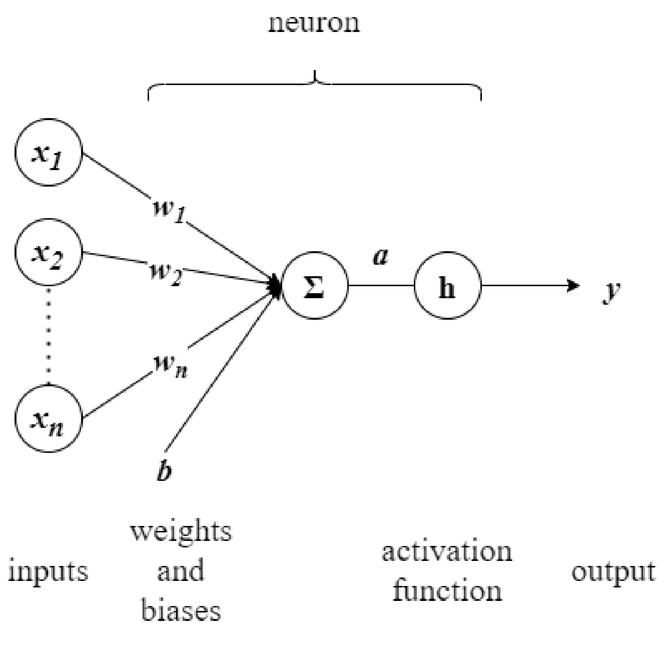


Figure 6: Neuron schema. The input is an n-dimensional vector x and the output is the prediction y. a is the neuron activity. The bias b can be treated as an additional weight w_{n+1} with input signal constant to 1 to simplify the notation. We adopt this notation hereinafter.

The activation function is used for providing non-linearity between inputs and outputs and it is the secret for the NN’s flexibility and expressive power; some examples of it include Sigmoid, rectified linear unit (ReLU) or hyperbolic tangent (tanh).

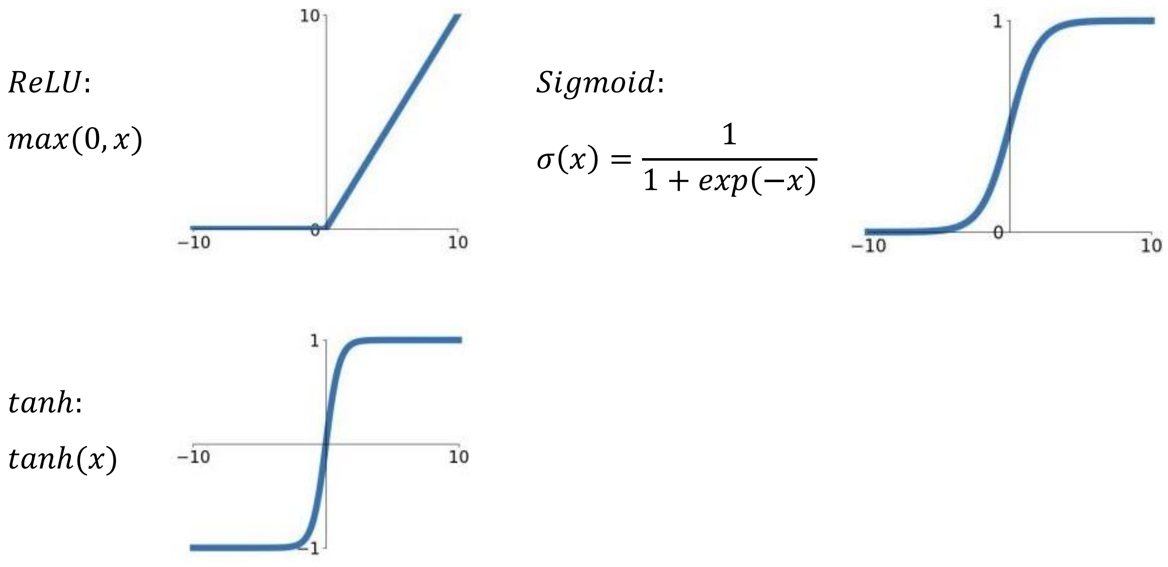


Figure 7: Commonly used activation functions: Sigmoid, ReLU, and tanh.

3.3.2 Feed-Forward Neural Network

The output of a neuron can become input to other neurons in a subsequent layer, resulting in the architecture of a deep network. A *feed-forward neural network*, sometimes called *Multi-Layer Perceptron (MLP)*, comprises neurons connected in an acyclic graph. Neural networks typically contain multiple layers of neurons: the first is called *input layer*, the last is the *output layer* and in the middle there are *hidden layers*. Neurons in middle layers are also called *hidden* or *latent features*, and they are a representation of the original vector in a latent space.

If x_i is the n-dimensional input vector for example i, $w^{(k)}$ represents weights in layer k and v are weights after last layer, the NN’s prediction $\hat{y}_i$ is given by:

$$\hat{y}_i = v^T h\left(W^{(2)} h\left(W^{(1)} x_i\right)\right) \tag{27}$$

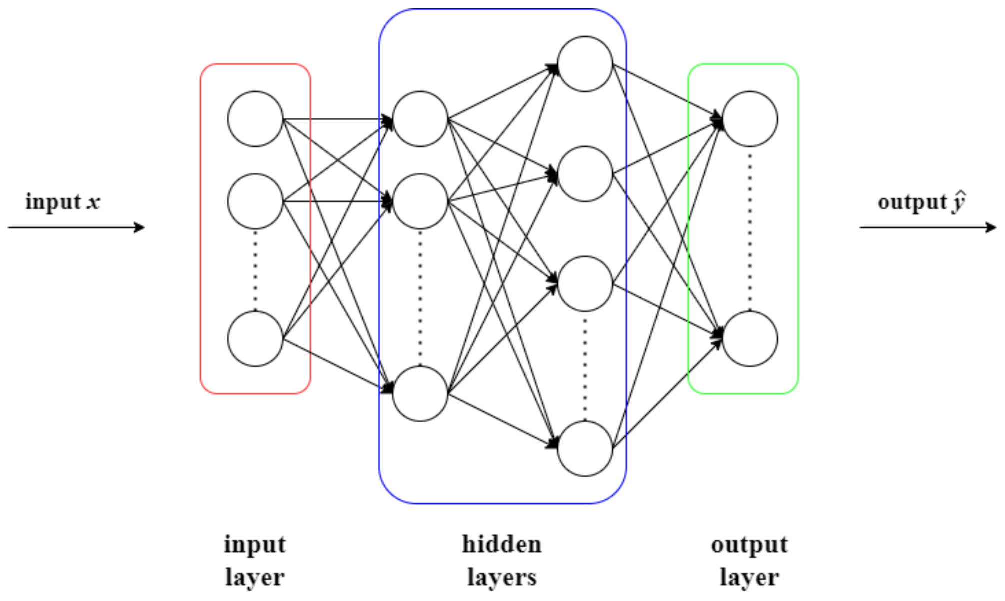


Figure 8: General high-level architecture of a multi-layer feed-forward neural network.

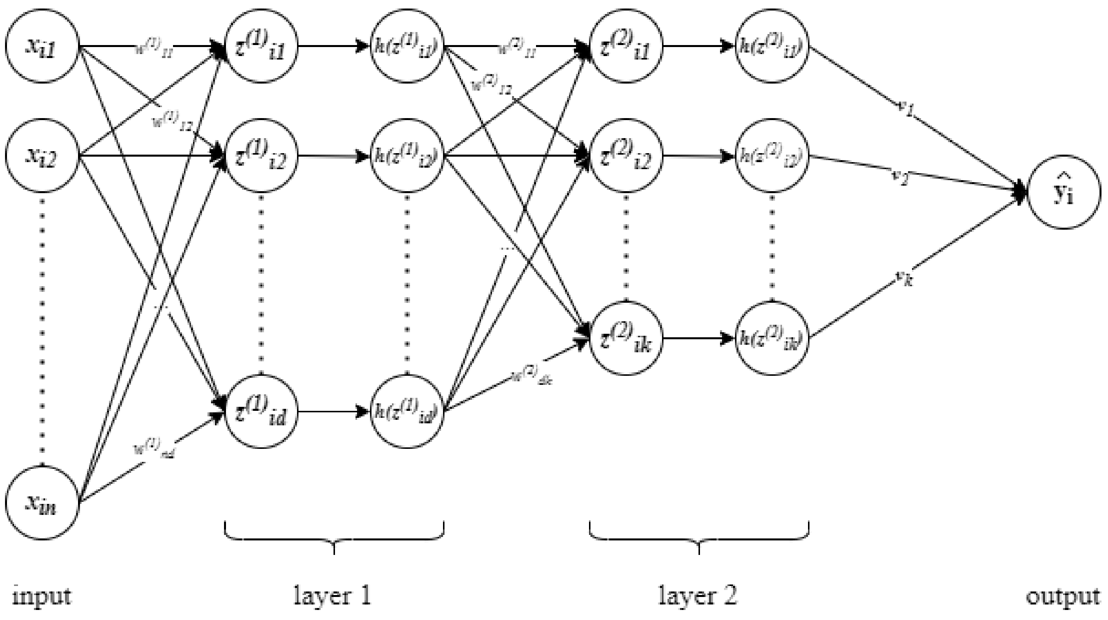


Figure 9*:* Two-layers neural network; all signals are detailed for the input sample i. Every circle represents a signal: input features x_i, hidden features $z_i^{(1)} = W^{(1)}x_i$ and $z_i^{(2)} = W^{(2)}h(W^{(1)}x_i)$, output $\hat{y}_i$. Although the output $\hat{y}_i$ is a scalar here, it might as well be a vector and, in that case, v is a matrix.

Non-linear activation functions between neurons are fundamental as the neuron itself performs a linear transformation, and if the activation is linear, then neuron and activation still represent a linear transformation. Adding non-linearities increases the expressive power of the model; it allows the NN to express more complex patterns in data compared to linear models such as *Support Vector Machines (SVMs)* [20].

The multiple-layered architecture is fundamental for the decomposition capability typical of Deep Learning models. A NN represents complex objects as hierarchical combination of re-useable parts (neurons), similarly to a simple grammar. Thanks to the network's architecture, neurons in shallow layers capture local properties while neurons in deep layers have a vision on broader patterns. For example, in an Optical Character Recognition (OCR) problem the input of the neural network is an image; each neuron recognizes a part of a digit, with shallow neurons recognizing small parts and deeper neurons recognizing combinations of parts.

Non-linearity elements and a sufficiently complex architecture allow feed-forward neural networks to be *universal approximators* [21] [22] [23], i.e. they can approximate arbitrarily well any well-behaved function.

3.3.3 Training a Neural Network

The training process a NN is based on a simple idea: if all structures are differentiable, weights are adjusted to reduce the prediction error. The training phase makes use of both the input and the desired output for the neural network, i.e. the feature vector and the target, as in a typical supervised learning setting. Training finds network parameters (weights and biases) that allow the NN to make predictions as close as possible to the desired target. A *loss function* is used to measure the error in prediction. Hence, training consists in learning weights and biases that reduce the network's loss.

ML models training ultimately consists in an optimization problem: finding the set of parameters that maximize or minimize an objective function – the loss or error. While DT training is a discrete optimization problem, NN is a typical example of continuous optimization. *Gradient descent (GD)* and its derivative methods [24] are the most popular class of training algorithms for continuous optimization in ML. GD is an iterative optimization algorithm; in order to reach the optimum point of the objective, it moves in the parameter space in the direction suggested by the gradient. Given a model with parameters w and loss function f, the approach is the following

- Start with a guess for model's parameters w^0.
- Successively refine the model's parameters at each iteration i:
    - Compute the gradient of f w.r.t. w for all training examples:

$$\nabla f\left(w^i\right) \tag{28}$$

    - Guided by the gradient, adjust w with the aim of obtaining the largest loss reduction:

$$w^{i+1} = w^i - \alpha^i \nabla f\left(w^i\right) \tag{29}$$

      α^i is called *step size* or *learning rate*. This computation decreases the value of f if the step size is small enough.
- Stop when a stopping criterion is met, usually threshold the gradient value:

$$\left|\left|\nabla f(w^t)\right|\right| \leq threshold \tag{30}$$

The objective function f must be differentiable. If it is convex and it admits optima, GD converges to a global optimum. *Stochastic Gradient Descent* (*SGD*) is a version of GD where the gradient is computed on a randomly-selected training example instead of all samples; SGD allows fast iterations with massive training sets. Most GD-derivative techniques are between GD and SGD: the parameters' update is performed by *batches*, i.e. groups of random training samples of intermediate size, allowing fast convergence a good solution quality with large datasets. *Adam* (*Adaptive moment estimation*) [25] focuses on learning rate scheduling and it is currently one of the most popular GD-based optimization algorithms; neural networks used in our experiments leverage this training method.

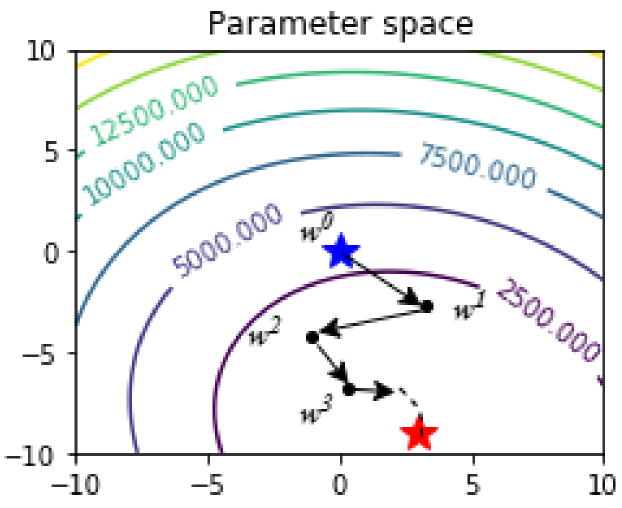


Figure 10*:* Gradient Descent in a two-dimension parameter space. The blue star is the starting point (i.e. the value of parameters at the beginning w^0), while the red star is the optimum. Arrows represent the descent performed in each GD iteration.

NN training requires the computation of the gradient of the loss with respect to the network's parameters, in order to incrementally adjust them to reduce the loss. *Backpropagation* is a technique to compute gradients in the NN and it is the base of gradient training for these models. It computes gradients via recursive application of the chain rule from calculus and it consists of two steps:

1. *Forward Propagation*. Training examples are fed to the neural network, computing the NN's output for them. With current and desired target values it is possible to compute the loss, i.e. how well current weights perform on samples.
2. *Backpropagation*: compute the gradient of the loss with respect to the weights.

After backpropagation Gradient Descent is applied, changing the NN's weights to reduce the error.

3.3.4 Neural Network Design

The NN's architecture must be handled properly in order to limit overfitting. The deeper is a neural network (i.e. more layers), the more complex and prone to overfitting it is. *Regularization* techniques are used to avoid overfitting, for example:

- *L2-regularization* or similar: add a penalty for large weights.
- *Early stopping*: stop GD training if the validation error does not improve.
- *Dropout*: randomly set some neurons to 0 on each GD training.

Neural networks are extremely powerful and flexible, but they must be carefully designed in order to obtain good performances. Several hyperparameters are involved in these models:

- Related to the NN architecture, for example number of layers, number of neurons per layer, activation functions, weights' initialization.
- Related to the GD algorithm, e.g. learning rate and batch size. GD-based algorithms are very sensitive to the learning rate, in particular for deep models: its value heavily affects convergence speed and model's performance, thus it must be carefully tuned.

Neural networks can be easily employed in both classification and regression tasks by adopting specific output layers and loss functions:

- Regression: no specific output layer is used, the predicted value $\hat{y}_i$ is the output of last neurons (it might also be a vector). Mean Squared Error or Mean Absolute Error can be used as loss functions. Formally, for a NN of 3 layers and MSE:

$$\hat{y}_i = v^T h\left(W^{(3)} h\left(W^{(2)} h\left(W^{(1)} x_i\right)\right)\right) \tag{31}$$

$$f\left(v, W^{(3)}, W^{(2)}, W^{(1)}\right) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{32}$$

- Classification: For multi-class or multi-label classification, v is a matrix and the output layer in the NN has one neuron for each class. The *softmax function* is used as activation function in the final layer to obtain a probability distribution over classes, i.e. probability value for each class*:*

  $$softmax\left(\hat{y}_{i,j}\right) = \frac{exp\left(\hat{y}_{i,j}\right)}{\sum_{c=1}^{k}\left(exp\left(\hat{y}_{i,c}\right)\right)}, j = 1, \dots k \tag{33}$$

  $$\hat{y}_i = softmax\left[v^T h\left(W^{(3)} h\left(W^{(2)} h\left(W^{(1)} x_i\right)\right)\right)\right] \tag{34}$$

  where there are k classes; $\hat{y}_i$ is a vector of k elements, one element $\hat{y}_{i,c}$ for each class c indicating the probability for sample i to belong to class c.
  Softmax loss, also called *cross-entropy loss,* is generally adopted as the error function. It measures the performance of a classification model whose output is a probability value:

  $$f\left(v, W^{(3)}, W^{(2)}, W^{(1)}\right) = \sum_{i=1}^{n}\left(-\hat{y}_{i,\widetilde{c_i}} + log\left(\sum_{c=1}^{k}\left(exp\left(\hat{y}_{i,c}\right)\right)\right)\right) \tag{35}$$

  where example i has true class $\widetilde{c_i}$.
  This activation function and loss is the general formulation. For mono-class and mono-label tasks these same functions are applied.

3.4 Additional ML Techniques

In sections 3.2 and 3.3 we reported the theoretical basis for decision trees and neural networks, the main machine learning methods under investigation in our system. Additional techniques are explored to model the relationships among variables but, for sake of brevity, we do not describe them in detail. They are used for exploration and they are not adopted in the combinatorial optimization model for the proposed system, thus a thorough technical explanation is not necessary. Nevertheless, we provide a brief introduction on additional ML models and techniques used in our experiments, together with references useful for a reader to gather further information.

3.4.1 Radial Basis Function

A *Radial Basis Function (RBF)* [26] is a function that depends on the distance between the input point and another point in the input domain. An example is Gaussian RBF, where the function is a gaussian, i.e.

$$gRBF(x, \tilde{x}) = exp\left(\frac{||\, x \; - \; \tilde{x}||^2}{2\sigma^2}\right) \tag{36}$$

where x is the input point and $\tilde{x}$ is the selected point in the domain. Gaussian RBFs are universal approximators, i.e. a sum of a large enough number of these functions can approximate any continuous function to arbitrary precision. RBFs are adopted in ML literature as basis in linear regression with modified basis, as kernels in SVMs and as activation functions in NNs.

3.4.2 K-Nearest Neighbors

K-nearest neighbors (*KNN*) [27] is a supervised ML technique. KNN assigns the target value to a new unseen point based on the values of its k closest examples from the training set. In classification, the assigned label is usually the most common label among the neighboring training points, i.e. the mode. In regression it usually the mean or the median among the neighbors' target values.

3.4.3 Linear Regression

Linear regression with polynomial or Gaussian RBF basis [28]. Linear regression is a statistical technique to approximate the relationship between a response and one or more explanatory variables with a linear model. With a non-linear feature transform (often called *change of basis*) it is possible to manipulate the features and represent polynomial or more complex relationships. Such a model in machine learning is trained by adjusting the model’s weights to minimize the prediction error.

3.4.4 Support Vector Machine

Support vector machines (*SVMs*) for classification and regression [20] [29] [30]. SVMs are a supervised learning technique; they represent the model that maximizes the margin between the training patterns and the decision boundary. For example, in classification tasks SVMs yield the maximum-margin classifier; this is the model that separates points in the input space with the hyperplane that is the farthest from all classes, i.e. longest distance to the closest examples in all classes. A change of basis allows such models to represent non-linear relationships. SVMs with L2 regularization and gaussian RBF [31] are extensively used in ML as they have fast training and testing, and they provide good performances as out-of-the-box models.

3.4.5 Principal Component Analysis

Principal component analysis (*PCA*) [32] is an unsupervised machine learning algorithm. It is a linear latent-factor model, i.e. a technique that simultaneously learns a change of basis from data and their representation in it. PCA represents the points of the dataset from their high-dimensional vector space to a different (usually lower-dimensional) space, by projecting them into the *principal components*, i.e. the directions of maximum variation for the data. Following the interpretation of PCA as a latent-factor model, principal components represent the new basis while the representation of data is the data themselves projected. PCA finds the low-dimensional hyper-plane minimizing variance in the projected space; it can also be viewed as the hyperplane minimizing the orthogonal distance to data. PCA is one of the most common techniques for dimensionality reduction and data visualization, and it is also adopted for data interpretation, outlier detection, and other applications.

Chapter 4
Empirical Model Learning

In the framework of a VPP's EMS, an additional combinatorial optimization problem can be introduced. Positioned on top of the hybrid online/offline algorithm, it has user-defined objectives and constraints. Its aim is to help the tuning of the online phase, e.g. choose number of traces in the contingency table, and to obtain forecasts about online performances.

The relationship between variables involved in this problem are modeled with machine learning. This allows to capture complex non-linear relationships with an automatic and data-driven approach. However, standard optimization tools are not designed to handle such situations, as the optimization problem involving these variables must model these special relationships. Namely, the modeling process must include these relations. Different approaches are reported in literature for leveraging ML to boost modeling of optimization problems [33]. The work proposed here focuses on Empirical Model Learning that allows embedding of ML models in the optimization model.

4.1 EML

A combinatorial optimization model encapsulates a description of the real-world system it deals with. This is usually a manually-defined model that describes only important aspects of the system, namely, making simplifications of the real world. A good high-fidelity model is fundamental for the success of the optimization process, and it needs to balance a tradeoff between complexity and accuracy: scarcity of details yields a poor description of the system with poor solution quality, whereas abundance of details leads to complex models with infeasible computational time. The modeling phase requires a thorough analysis of the system itself with the help of domain experts.

Although literature exposes methods to address modeling for real-world systems, high-complexity systems still represent a challenge for optimization problems due to modeling. *Empirical Model Learning* (*EML*) [34] [3] merges the two fields of machine learning and optimization in this exact direction: it leverages ML to tame high-complexity systems modeling, making such systems treatable in combinatorial optimization.

EML is a technique that allows combinatorial optimization in highly complex systems. It handles optimization for complex real-world systems in two steps, adopting a similar approach to the one followed in our work:

- Given a system that is hard to model by conventional means, use a machine learning model (*Empirical Model*) to approximate its input/output behavior, i.e. the relations between variables.
- Encapsulate these relationships into components of a combinatorial optimization problem, i.e. constraints and objectives. In other words, *embed* trained ML models into the optimization problem.

The technique proposed in [34] is focused on the modeling side of the optimization problem and it digs into methods for performing the embedding, with emphasis on specific ML models (decision trees, random forests and artificial neural networks) and several optimization techniques (local search, mixed integer linear programming, constraint programming, SAT modulo theories). A well-designed modeling of empirical models is fundamental for the entire optimization process: embeddings should be designed so that the optimization engine can exploit the empirical model's structure for boosting the search operation. Hence, embedding is not just a matter of encoding ML models, because it must address the efficient use of optimization components – variables, constraints, and objectives.

[3] *https://emlopt.github.io*

An EML-based system is capable of suggesting optimal decisions in a highly-complex real-world setting. EML allows the integration of advanced predictive modeling and big data analysis techniques into prescriptive analytics, by virtue of its ability to integrate machine learning into combinatorial optimization. EML can be interpreted as a technique to merge predictive and prescriptive analytics.

The approach stemming from EML is data-driven and automatic. Compared to classic optimization modeling it offers a new vision where part of the prescriptive model is extracted from data; EML-based models are crafted on real-world data because they rely on empirical machine learning models to obtain components of the prescriptive optimization problem.

EML introduces approximations in its embeddings that are necessary to allow usefulness of the techniques: "*In EML, models are useful if they provide adequate accuracy, and if they can be effectively exploited by solvers for finding high-quality solutions*" [34]. However, this is not peculiar of EML: models in both predictive and prescriptive analysis introduce some forms of approximation; these are inevitable to enable the applicability of the techniques. machine learning models are based on statistics and approximation is fundamental to allow their generalization capability. Also classic optimization techniques have approximations: usual human-defined optimization models are approximations of the real system, and this is the key for modeling computationally-feasible problems.

The Empirical Model Learning approach enables the application of optimization techniques to complex real-world problems that used to be hard to tackle. Additionally, it easies the need for hand-crafted models by domain experts and it opens up new application areas. Designing a good empirical model may still be a non-trivial task, but EML allows better accuracy compared to manually-crafted expert-designed approaches. Experiments in [34] show the clear advantages of using a data-extracted model in terms of quality of the final solutions.

4.2 Optimization Problem Modeling

Combinatorial optimization problems handled with EML can be formulated as a set of variables, constraints, and objectives. Assume z is the vector of observables, x is the vector of decision variables x_i, each with domain D_i, f is the cost function to optimize and g_j are predicates representing constraints, e.g. inequalities. These are typical components of a combinatorial optimization problem, referred to as the *core combinatorial structure*. EML brings an additional part into this model: a function h representing the embedding of the empirical machine learning models. It specifies how the observables z depend on the decision variables x, hence providing an approximate description of the behavior of the high-complexity system.

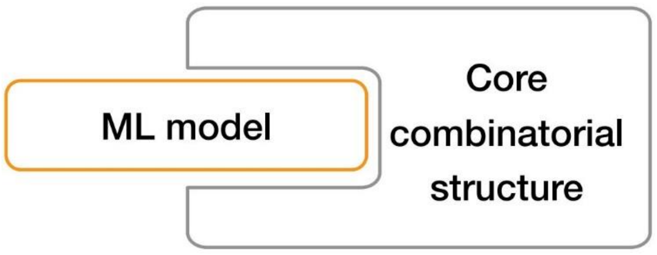


Figure 11*:* In EML, the optimization problem is composed of the (original) core combinatorial structure and a empirical machine learning model. From: M. Lombardi, M. Milano, EML repository [4]

A general optimization problem modeled with EML has the following formulation, where (37), (38) and, (40) are the core combinatorial structure and (39) is the EML-specific contribution:

$$min\, f(x, z) \tag{37}$$

$$g_j(x, z), \forall j \in J \tag{38}$$

$$z = h(x) \tag{39}$$

$$x_i \in D_i \;, \forall\, x_i \in x \tag{40}$$

[4] *https://emlopt.github.io/assets/*

Modeling such combinatorial optimization problem includes three main steps:

1. Define the core combinatorial structure of the problem. This phase consists in defining a combinatorial optimization model as in usual optimization workflow.
2. Obtain the empirical model, i.e. train machine learning models involving variables of interest for either regression or classification. This step includes the classic machine learning model construction process, as described in section 3.1.1.
3. Embed the empirical model in the combinatorial optimization problem. This step is peculiar of EML and it represents its core. Embedding is possible if and only if the ML models are associated to an encoding for the optimization problem adopted. EML defines these encodings, with a design that allows the encoded ML model to be exploited by the optimization approach for boosting the search process.

At the end of these steps the problem is entirely modeled, and it can be solved with a usual optimization model resolution technique.

4.3 Empirical Model Embedding

Embedding a trained machine learning model into an optimization problem requires to define encodings for the ML model. They translate the empirical model into custom variables and constraints to be integrated inside the core combinatorial structure.

The integration also requires defining an *operational semantics*, whose goal is to allow the efficient use of encoded components. Operational semantic refers to any procedure that helps the optimization engine to boost the search process by reasoning over the specific empirical ML model, e.g. exploiting bound computation or constraint propagation. It may be provided implicitly by the underlying solver or in some cases it is defined explicitly together with the encoding, by adopting specific algorithms and components in the optimization problem. In a MILP problem for example, once the

ML model has been embedded its equations are automatically taken it into account by the solver for computing bounds and generating cuts, thus improving the search phase.

EML presents embedding techniques for artificial neural networks, decision trees, and random forests and four combinatorial optimization approaches, namely, local search (LS), mixed (non) integer linear programming (MILP), constraint programming (CP), and SAT modulo theories (SMT). Our experiments will focus specifically on NNs and DTs as empirical models, and MILP as optimization technique.

4.3.1 Decision Trees

Decision trees and random forest embedding into a combinatorial optimization problem is introduced in [35].

To embed a DT, in the first step all the necessary variables are introduced into the model: a decision variable for each input attribute, assume x is a vector representing input features, and a decision variable for the class y. Categorical attributes and classes can be modeled as integer variables.

Then, consistency on the relationship given by the DT is modeled and enforced:

$$y = decision\ tree(x) \tag{41}$$

A simple encoding follows the intuition that a tree defines several paths, and each path is an implication. Hence, a DT can be then encoded as a set of constraints that represent boolean predicates for its paths. Assume π is a path from root to leaf in the tree, $C(\pi)$ is the class corresponding to the leaf in the path π and each $b_j \in \pi$ is a branch along the path. Each expression $cst\left(b_j\right)$ represents the condition in the branch b_j and is given by:

$$cst(b_j) = \begin{cases} \bigvee_{v \in L(b_j)} [\![x(b_j) = v]\!] & if\ x(b_j)\ is\ symbolic \\ [\![x(b_j) \leq t(b_j)]\!] & if\ x(b_j)\ is\ numeric\ and\ b_j\ is\ a\ left-branch \\ [\![x(b_j) > t(b_j)]\!] & if\ x(b_j)\ is\ numeric\ and\ b_j\ is\ a\ right-branch \end{cases} \tag{42}$$

where $[\![*]\!]$ denotes the truth value of the boolean predicate/constraint $*$, and $x(b_j)$ is the attribute variable tested in branch b_j. A simple rule-based encoding is obtained following the observation that each path π from root to leaf can be interpreted as a logical implication that includes all branches along the path:

$$\bigwedge_{b_j \in \pi} cst(b_j) \Rightarrow [\![y = C(\pi)]\!]\ , \pi \in paths \tag{43}$$

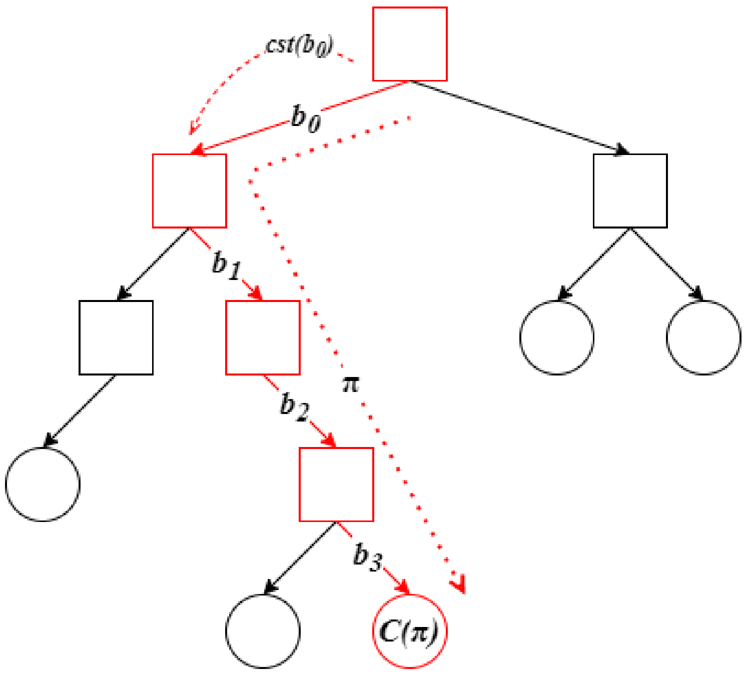


Figure 12: Representation of a DT in EML. A path π in the DT is represented by a logical implication involving all conditions $cst(b_j)$ along the path, leading to the label $C(\pi)$.

This expression applied to each path in the tree is sufficient to encode a DT.

However, it is possible to obtain a formulation of the encoding that yields a stronger propagation. The key observation is that the set of leaves labeled with a certain class specifies all and only the input configurations that should be associated to such class. The class variable y takes the value c if and only if at least one of the implications associated to the paths π_c labeled with c is true. This allows one to encode an entire tree as a set of clauses, formally:

$$\forall\, c \in classes :\ [\![y = c]\!] \Leftrightarrow \bigvee_{\pi_c:\, C(\pi_c) = c} \left[\bigwedge_{b_j \in \pi_c} cst(b_j) \right] \tag{44}$$

If logical constraints are not supported by the optimization approach, it is possible to obtain a formulation equivalent to Eq. (44); to do that, the left-to-right and right-to-left implications associated with the biconditional operator (⇔) are separately modeled. In particular, the right-to-left implication corresponds to Eq. (43) and for the whole DT it translates to:

$$\forall\, \pi \in paths :\ \prod_{b_j \in \pi} cst(b_j) \leq [\![y = C(\pi)]\!] \tag{45}$$

where $[\![*]\!]$ is 1 if the logical expression * is true and 0 if it is false. Intuitively, this equation forces the class variable y to take the value $C(\pi)$ if the current values of the attribute variables are such that all the $cst(b_j)$ constraints are satisfied. On the other hand, if y takes the value $C(\pi)$, then at least one of the conjunctions of $cst(b_j)$ constraints associated to that class must be true. This leads to the formulation for the left-to-right implication of Eq. (44) as:

$$\forall\, c \in classes :\ [\![y = c]\!] \leq \sum_{\pi_c:\, C(\pi_c) = c} \left[\prod_{b_j \in \pi_c} cst(b_j) \right] \tag{46}$$

In an optimization problem formulated as a MILP, the decision tree is embedded according to the following steps:

1. Obtain the decision tree in rule format, transforming the DT into a set of rules. Every rule represents a path from the root to a leaf. Each rule is composed of attribute name, attribute type, and threshold test needed to go on along the path. The last element of the rule represents the class label of the leaf.
2. Introduce a binary variable for each rule. A constraint is used to enforce that only one rule can be active at a time.
3. Process all conditions in all rules to maximize efficiency, e.g. collapse conditions on the same attribute for each rule.

4.3.2 Neural Networks

Additional works in literature besides EML examine how neural networks can be efficiently employed in combinatorial optimization problems [36]. A neural network is a declarative non-linear model. It can be embedded in a combinatorial optimization model directly by inserting variables for its inputs and outputs in the model, then introducing the NN's equations in the model.

To embed a NN, variables to model inputs and outputs for each neuron are introduced. Then, the neuron's equations are directly inserted into the model; each neuron is represented in the combinatorial problem by a variable and a *neuron constraint,* that ties the neuron's input and outputs. In a network where several neurons are combined, each edge is modeled as a constraint on the connected neurons.

$$\begin{cases} a = b + w^T x \\ y = h(a) \end{cases} \tag{47}$$

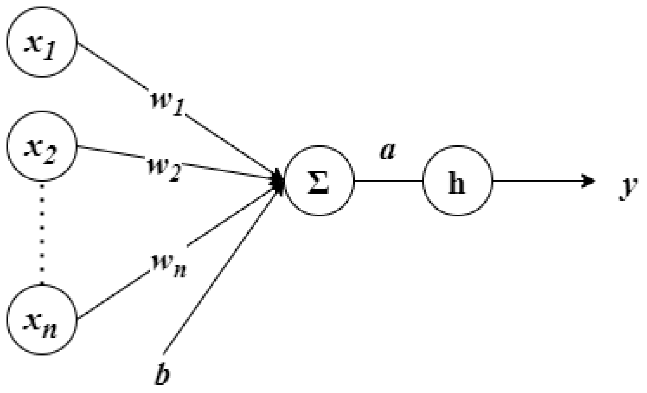


Figure 13*:* Neuron schema in EML, adopting a similar notation to section 3.

The embedding is straightforward as long as the optimizer supports the activation functions adopted in the neurons. If the activation is not directly implemented by the solver, workarounds can be adopted to yield them. For example, indicator constraints and a slack variable are used to build ReLU activations in MILP problems [37].

However, there are some aspects that must be considered when neural networks are embedded, as reported in [37]. In NNs with several hidden layers and neurons that use non-linear functions, the cost function might be not convex [38]. Some solvers rely on convexity for providing globally optimal results; in this case the solver would converge to a local optimum, possibly different from the global optimum. Another potential problem is numerical stability in the resolution process. Some MINLP solvers, for example, perform inversion on the model’s functions in some of their resolution steps. Even when an activation function is invertible, due to the finite precision of the underlying machine, inversion may be possible only within a restriction of the function’s domain; this might lead to loss of some feasible solutions or even software crashes. The issue can be addressed by restricting domains of the input/output variables of each neuron; according to [37], in rare cases this process might accidentally eliminate a high-quality solution.

Chapter 5
Problem and Implementation

In a VPP optimization problem, the conventional optimizer decides power flows in the grid. Assume this component adopts the hybrid online/offline approach for performing optimization under uncertainty, leveraging the fixing heuristic and a contingency table. Design choices must be taken on this component to balance the tradeoff between solution quality and online computational resources. The algorithm can be tuned to achieve a good solution quality or to satisfy time/memory constraints, depending on the requirements in a specific situation.

We propose a high-level optimizer that incorporates the behavior of both the controlled system, i.e. the VPP, and the online/offline optimizer. Inserted in a multi-level hierarchical optimization system, this new component is used to guide configuration decisions or perform forecasts on the low-level online optimization process. machine learning models are used to capture the complex behavior of the VPP/hybrid optimizer system. The high-level optimizer leverages EML to incorporate them into its combinatorial optimization model.

This section focuses on the design of the high-level optimizer. After an introduction on the multi-level hierarchical optimization system, the machine learning models are presented, and finally the proposed optimizer is detailed.

The system's construction (both ML models and optimization problems) and all experiments are performed on cloud with the Google Colab[5] platform. This allows us to leverage larger computational resources compared to local machines and to facilitate shareability of models, results, and code. The main tool adopted for training and testing

[5] *https://colab.research.google.com/*

machine learning models is scikit-learn[6], although neural networks are modeled in Keras[7] for TensorFlow[8]. IBM cplex[9] is used as solver for the combinatorial optimization problem. For all these tools we used the offered Python APIs.

The construction of ML models, the modeling of the high-level optimizer and all the experiments are performed using data already produced by the hybrid offline/online optimizer as detailed in section 2.3.7.

5.1 System

The complete optimization system proposed in this work for the virtual power plant is structured as a hierarchy of optimizers:

1. *Low-level optimizer*. Optimization under uncertainty is performed adopting the hybrid offline/online approach. Specifically, the online step of this component leverages the fixing heuristic and the offline-computed contingency table, as described in section 2.3.5. This is the real VPP optimization, namely, deciding power flows based on the objective (cost minimization) and based on the stochastic factors.
2. *High-level optimizer*. The high-level optimizer performs decision-making at a higher level, on top of the first optimizer. It incorporates the behavior of both the controlled VPP system and the low-level optimizer, learned through machine learning models. It does not have the same view of low-level optimizers, i.e. all power flows. However, it handles factors involved in the low-level stochastic optimization, e.g. number of traces for the contingency table, online optimizer’s computational cost. This optimizer is highly flexible:

[6] *https://scikit-learn.org*

[7] *https://keras.io*

[8] *https://www.tensorflow.org*

[9] *https://www.ibm.com/analytics/cplex-optimizer*

it is customizable to allow the definition of the desired constraints and objective, and it can be easily used for either deciding the low-level optimizer's configuration or forecasting its behavior.

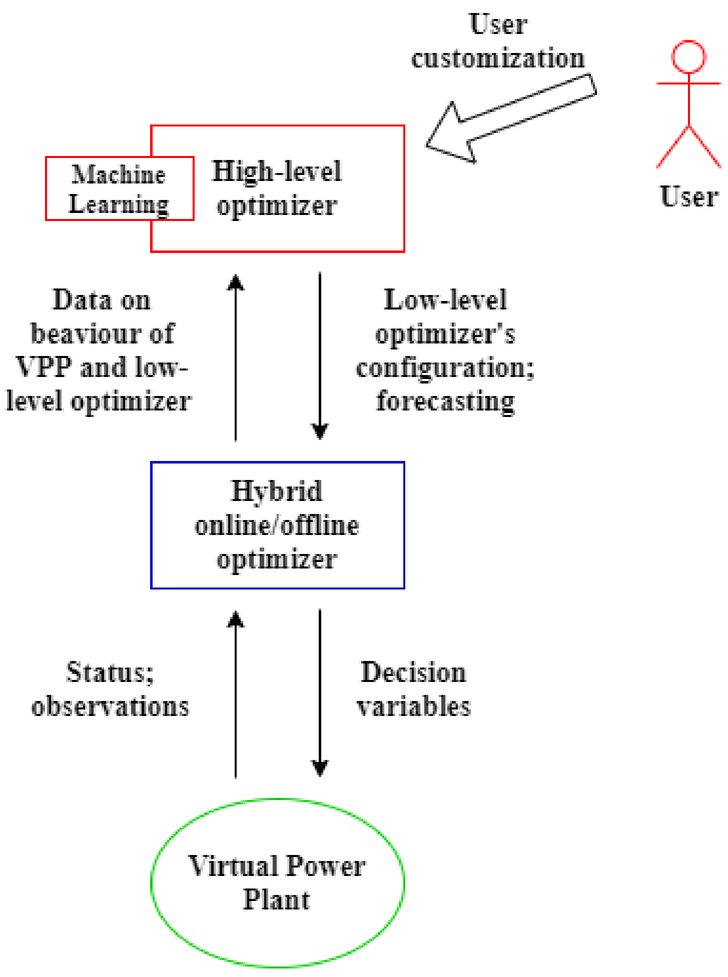

User customization
User
Machine Learning
High-level optimizer
Data on beaviour of VPP and low-level optimizer
Low-level optimizer's configuration; forecasting
Hybrid online/offline optimizer
Status; observations
Decision variables
Virtual Power Plant

Figure 14: General overview of the entire VPP optimization system. The low-level hybrid offline/online optimizer performs stochastic optimization for the VPP. The high-level optimizer allows both decision-making on the configuration and performance forecasting for the low-level optimizer; it is data-driven, flexible, and customizable by the user.

5.2 Dataset Analysis

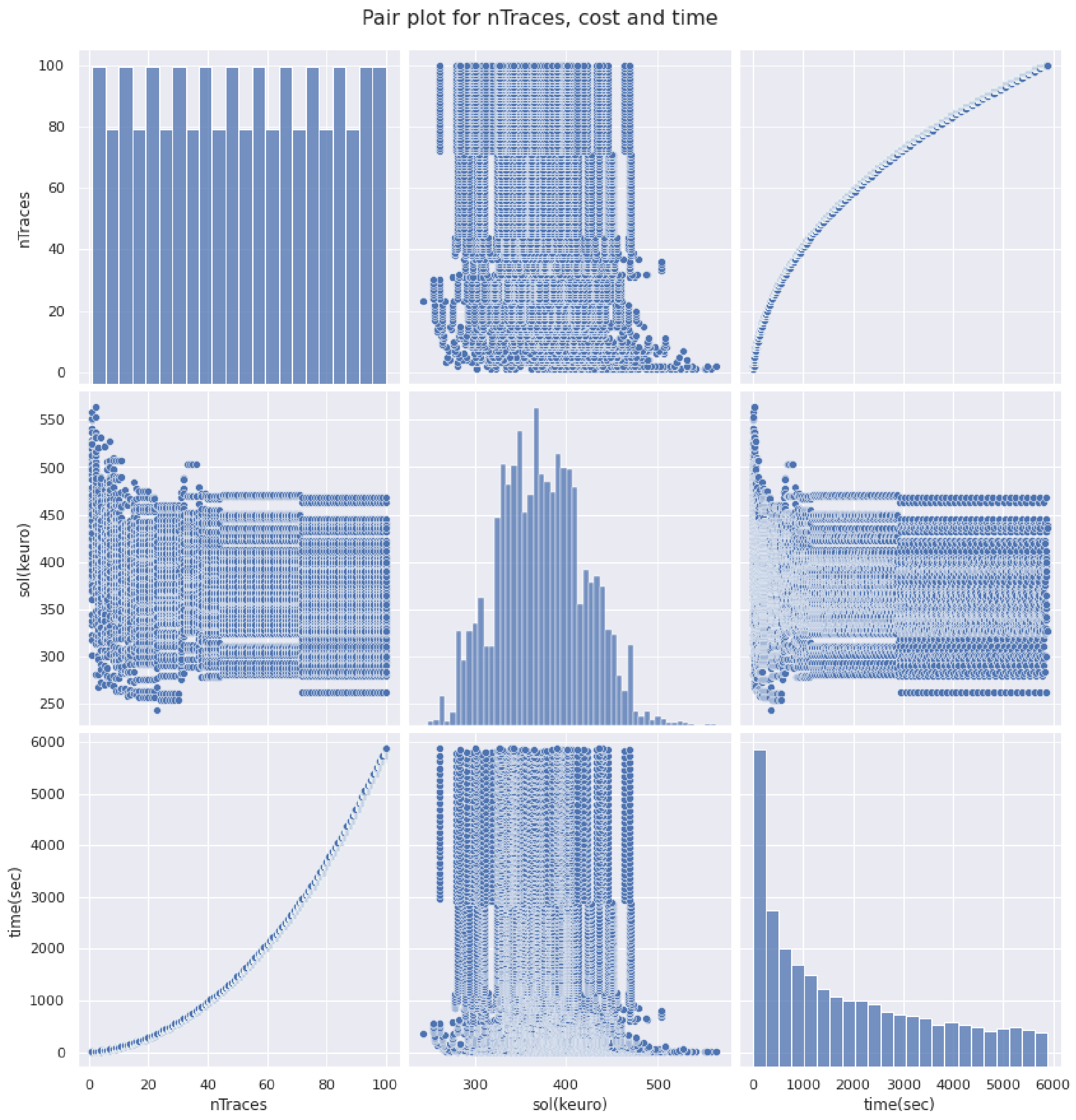
Before building models, we perform a thorough data analysis on the dataset. This phase is fundamental for understanding the quality of the dataset and deciding whether data cleaning or preprocessing is necessary before building ML models. It also allows to capture evident patterns or relations and it gives information on which ML approaches might be suitable for representing them.

Data used to construct the high-level optimizer concerns the behavior of the low-level hybrid offline/online optimizer that leverages the fixing heuristic and a contingency table computed offline, i.e. CONTINGENCY in section 2.3.5. The dataset is the result of several runs of this algorithm, with different parameters and on different problem instances. An *instance* is the data optimized by the low-level optimizer, i.e. a sequence of realizations of the stochastic variables of the VPP. Specifically, the dataset is generated with 100 different instances. Each instance is fed as input to the hybrid optimization approach (with fixing heuristic) 100 times, for a varying number of traces in the contingency table T from 1 to 100. As a consequence, the dataset is composed of 100 x 100 = 10000 entries. The features are data associated to each run of the hybrid optimizer; there are information on its input (i.e. data regarding the instance), its configuration, the time and computational resources required for the online optimization, and the solution:

- Information about the instance
  - PV and $Load$. Sequence of realization for the variables PV and $Load$ in all stages. These features represent information on the instance. Each of them is a vector of 96 values.
- Configuration of the hybrid offline/online optimizer
  - $nTraces = |T|$. Number of traces in the contingency table T used by the fixing heuristic in that run.
- Solution found by the hybrid offline/online optimizer

    - *Cost*. Solution value, i.e. cost, found by the hybrid offline/online optimizer, in kEuros. Lower cost indicates a better solution quality.
- Information on run-time and resources required by the online computation of the hybrid optimizer
    - *CostNorm*. Solution cost, normalized to the baseline.
    - *Time*. Time required by the hybrid offline/online optimizer for the online computation, in seconds.
    - *TimeNorm*. Time, normalized to the baseline.
    - *AvgMem*. Average memory used by the hybrid offline/online optimizer during the online computation, in MB.
    - *AvgMemNorm*. Average memory, normalized to the baseline.
    - *MaxMem*. Maximum memory used by the hybrid offline/online optimizer during the online computation, in MB.
    - *AvgCPU*. Average CPU amount used by the hybrid offline/online optimizer during the online computation, in % of used CPU.
    - *AvgCPUNorm*. Average CPU amount, normalized to the baseline.
    - *MaxCPU*. Maximum CPU amount used by the hybrid offline/online optimizer during the online computation, in % of used CPU.

We plot pairwise relationships between all variables relevant in our models. In the plot grid, when two variables are involved the scatterplot is reported. On the other hand, when one variable is plotted against itself (i.e. diagonal of the grid), the univariate distribution plot is drawn; it shows the marginal distribution for the variable. These plots give information about the distribution and the relationships between variables; they are fundamental when building machine learning models as they might suggest the adoption of specific techniques and help us understand why some models have a good (or poor) performance.



Graph 1: Pair plot for number of traces, solution cost, and time.

According to the plots above:

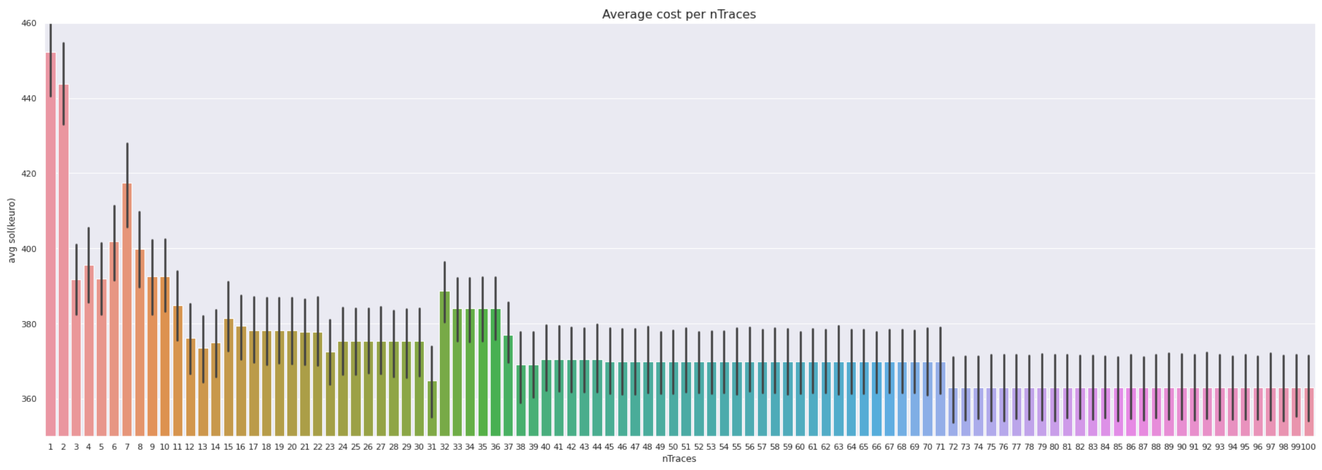
- There is a strong correlation between number of traces and resolution time, apparently a quadratic or low-degree polynomial relationship. Namely, time increases as the number of traces grows, and this trend is present in all instances.

- The solution cost does not have such a neat correlation with the number of traces. For low values of nTraces, cost decreases as the traces increase. On the other hand, for a high number of traces (after approximately 40 nTraces) it is constant. This is not surprising because for small amounts of nTraces adding new traces to guide the fixing heuristic helps the solution quality; when their amount is already large, the cost does not benefit from adding traces.
- Similar correlation between time and solution cost. For low values of time the cost goes down as the time grows, whereas cost is constant when time changes for larger values of time. However, for small values of time the correlation between cost and time is less neat compared to the one with nTraces. It is worth to mention that this relationship might be indirect, i.e. due to the fact that time is strongly correlated to nTraces and nTraces is to cost.
- The solution cost follows a gaussian distribution, according to its marginal distribution. Time on the other hand is more present for small values and its distribution decreases as its value increases.

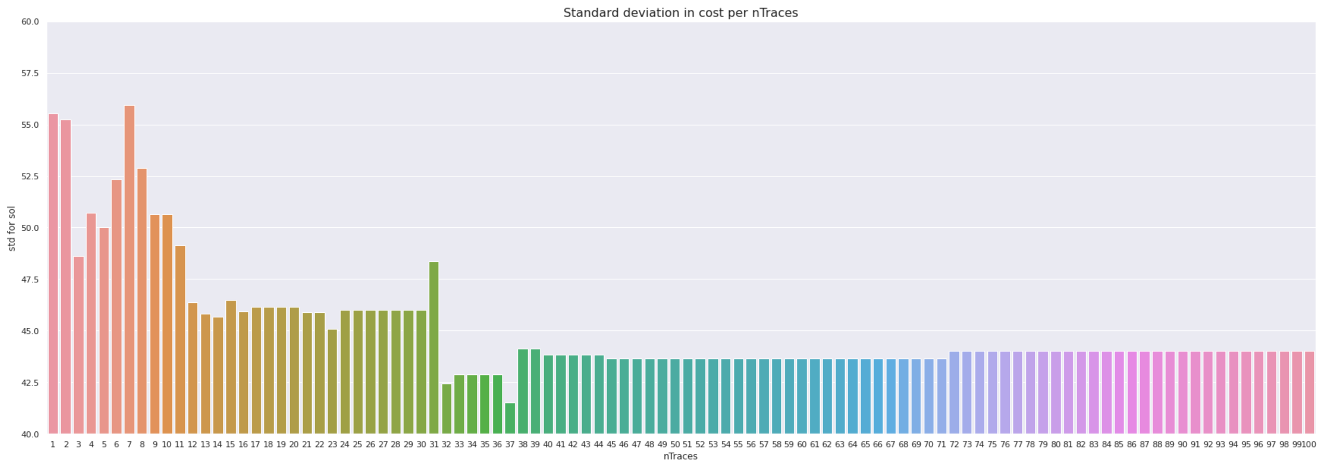
Additional plots are reported in Appendix A help to shed light on the relationship between the variables. In particular:

- It is confirmed that the correlation between time and cost is a consequence of the strong relationships between time-nTraces and nTraces-cost.
- From the marginal distribution of cost with number of traces colored, we see that the probability density of cost if we fix nTraces is a gaussian that is higher and thinner for large values of nTraces.
- Correlation between nTraces and time is very clear, and it is not influenced by cost. This strong correlation can be exploited to predict nTraces given the time, as a quadratic or polynomial model would fit it perfectly.

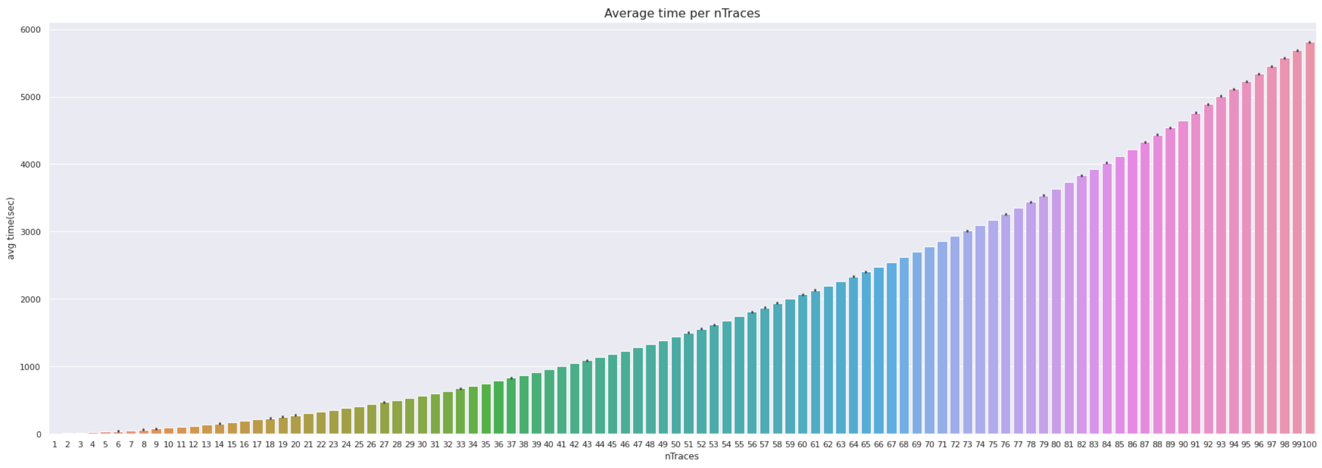
We plot below the dataset with bar charts showing average and variance values for solution cost and resolution time across all 100 instances, varying the number of traces. It is important to explore these relationships because they represent the most important parts of the behavior that the high-level optimization model must capture.



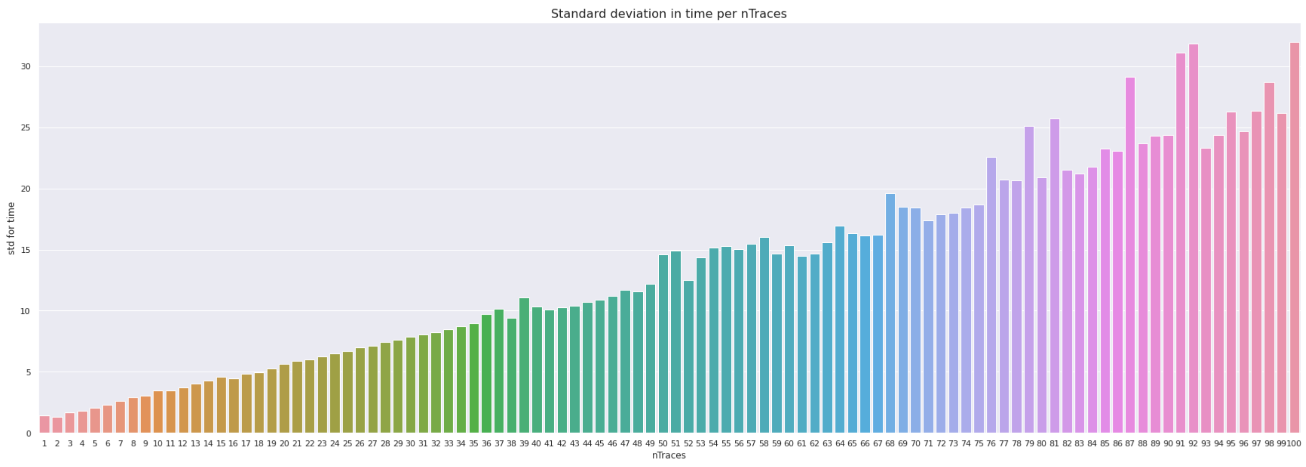
Graph 2: Average solution cost across all 100 instances, for each value of the number of traces. The variance is also reported in each bar.



Graph 3: Standard deviation for the solution cost across all 100 instances, for each value of the number of traces.



Graph 4: Average online resolution time across all 100 instances, for each value of the number of traces. The variance is also reported in each bar.



Graph 5*:* Standard deviation for the online resolution time across all 100 instances, for each value of the number of traces.

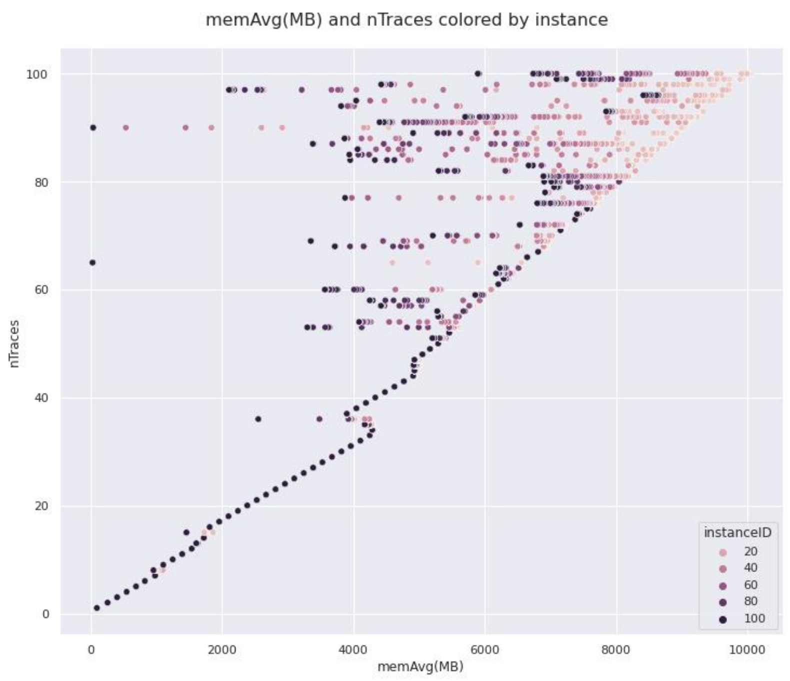
These plots confirm the previous considerations about the relationship between number of traces and solution cost. For low number of traces the cost is influenced by the number of traces, namely, more traces yield a better solution quality. On the other hand, after 40 traces the solution cost is approximately constant and nTraces does not influence it. The variability of solution cost for all instances is also higher for low number of traces compared to a large nTraces.

The computational time is strongly correlated to number of traces: as nTraces grows the time grows, following a low-degree polynomial relationship. The variance increases steadily as nTraces grows.

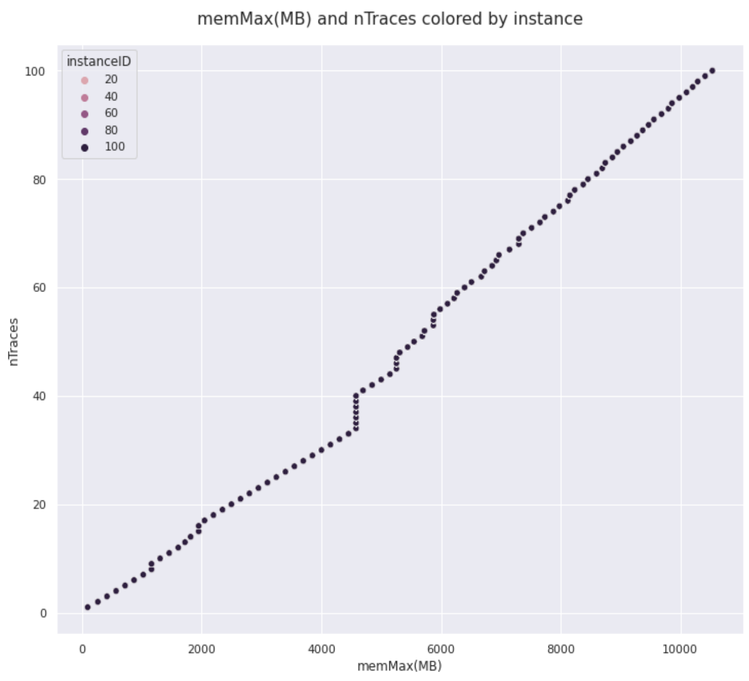
We leverage scatter plots to examine the relationship between number of traces and memory (either average or maximum) used by the hybrid optimizer in the online computation. We also color the instance id to examine how this relationship is influenced by characteristics of each instance. Plots are reported below.

According to the graphs, maximum memory and nTraces have a positive linear relationship that is strong and is not influenced by the instance.

The relationship between nTraces and average memory is more complex. For small amounts of traces, all instances have the same value of average memory, and there is a linear positive correlation between average memory and nTraces. On the other hand, for large values of nTraces the relation is less neat. For some instances there is still a positive linear correlation between the two variables, whereas for other instances the relationship is more chaotic and increasing nTraces leads to lower average memory.



Graph 6: Scatterplot between average memory and number of traces. The instance id is colored.



Graph 7: Scatterplot between maximum memory and number of traces. The instance id is colored.

5.3 Machine Learning Models

We construct ML models that capture the behavior of the low-level components in the system, i.e. controlled VPP and offline/online optimizer.

Specifically, variables involved are: number of traces (nTraces), computational time and average memory of the online algorithm, solution cost, and two variables related to the instance, i.e. the sequence of PV and Load values. There are three relationships among variables that are relevant to model:

- Relationship between nTraces and computational time;

- Relationship between nTraces and solution cost;
- Relationship between nTraces and given memory.

They can be modeled by either taking or not taking into account information on the instance, namely, PV and Load.

The number of traces is an integer between 1 and 100 and can be considered a categorical value. However, the best models that predict this target treat it as numerical value, i.e. regression instead of classification. Regressors will be adopted, and constraints enforcing nTraces to be integer can be applied later in the optimization model.

All models are built and evaluated by performing a 0.8/0.2 train/test random split, i.e. 80% of randomly selected samples from the dataset are used for training and the remaining 20% for testing.

We evaluate the performance of models in terms of:

- *Accuracy* on the test set for classification tasks. The accuracy score reports the number of correct class predictions across all samples. Assume there are n test samples, $\hat{y}_i$ is the predicted label for sample i and y_i is its *ground truth* (i.e. real) label; then the accuracy score is:

$$accuracy = \frac{1}{n}\sum_{i=1}^{n} [\![y_i = \hat{y}_i]\!] \tag{48}$$

  where $[\![*]\!]$ is 1 if the predicate * is true, 0 otherwise. Values are between 0 and 1, larger values indicate better the performance.
- *R-Squared* (R^2 or *coefficient of determination*) score on the test set for regression tasks. Assume $\hat{y}_i$ is the predicted value for sample i and y_i is its *ground truth* (i.e. real value), there are n test samples, μ is the mean value of y across all samples; then R^2 is:

$$\mu = \frac{1}{n}\sum_{i=1}^{n} y_i \tag{49}$$

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \mu)^2} \tag{50}$$

A larger value indicates better the performance; the best possible R^2 score is 1.0 and it can assume negative values.

It is worth to note that the numerator in (50) is the *sum of squared errors* (*SSE*):

$$SSE = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2 = \sum_{i=1}^{n}(\varepsilon_i)^2 \tag{51}$$

and it has the same formulation of the MSE in (23) except for a constant. Hence, R^2 has a correspondence to a prediction error measure: low values of this score indicate large errors whereas the largest value (i.e. $R^2 = 1$) correspond to an error of 0.

5.3.1 All Models

Although adopting EML to integrate empirical models with the optimization problem limits the available types of ML models, we perform a preliminary analysis in a general way. We explore several state-of-the-art machine learning techniques, and we do not limit to the models used in EML, namely, decision trees and neural network. With this approach we compare the final ML techniques to different ones; this allows us to understand whether the final empirical models used in the optimization problem perform a good modeling of data, or other ML methods outperform them. This phase is also important to explore and guide the design of empirical models that will be employed in our system.

5.3.1.1 ML Models Exploration

We leverage different machine learning techniques to capture the relationships between two or more variables. For the same set of variables and targets we build several models with multiple ML techniques and we compare their performance.

We report here, for sake of brevity, details and results only for the best models obtained with each ML method. Complete results with additional models are reported in Appendix B.

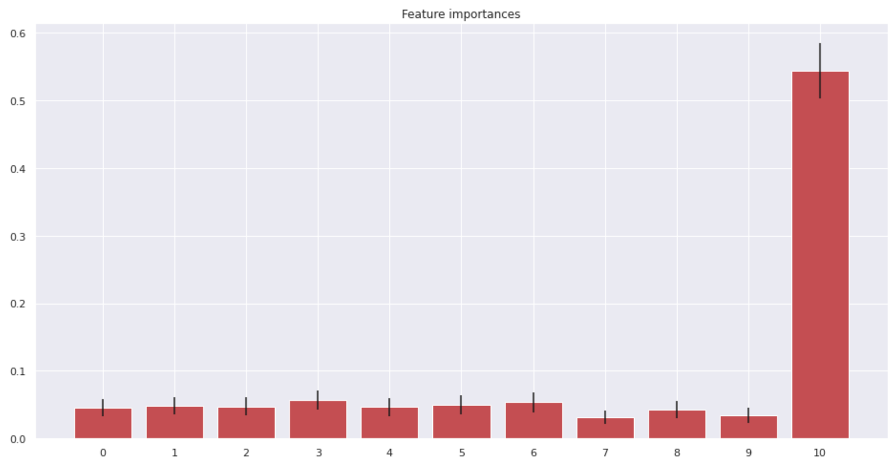
i. nTraces and cost

Focus on the relationship between number of traces and solution value. Each model predicts nTraces and uses as features either cost, PV and load or just cost. Plots in Appendix A help to shed light on the relation between these two variables.

The best model found for each ML technique are:

- Linear regression with change of basis to quadratic base, i.e. polynomial base degree 2. Cost, PV, and load are used as features.
- Extra Trees regressor is a random forest with additional randomness. In each node in the tree the split threshold is selected among a set of values randomly chosen, instead of taking the best value possible, yielding regularization effects. We use two trained random forest classifier (max depth 6, num trees 5) to generate a “feature encoding” for PV and Load. This process yields an informed (i.e. trained) representation of PV and Load; it works as dimensionality reduction for the two vectors, each going from 96 to 5 dimension. It brings a slight improvement compared to the original 96-dimensional representation. Then, use Extra Trees to perform regression taking as features the cost and the two reduced PV and Load vectors.

- Visualization of feature importance for the best RF model:



Graph 8*:* Feature importance for the RF. Features 0-4 are PV, 5-9 are Load, and 10 is cost.

- Using only the cost as feature (i.e., no PV and load) results in a slight degradation: 0.85111 vs 0.89015 R^2 score. Remarkably, cost alone already achieves a high score; adding PV and Load as predictors improves the performnce but not by a large amount.

- NN regression. Architecture [400], ReLU as activation function, Adam with learning rate 0.001, no validation-based early stopping, batch size 200, 500 epochs. Cost, PV and load are used as features.
- Support Vector Machine regressor with Radial Basis Function kernel and regularization parameter C=100. Cost, PV and load are used as features.

Model	**LR**	**RF**	**NN**	**SVM**
R^2	0.28743	0.89077	0.07606	0.90514

Table 1: Test set performance for the best regressors that predict nTraces using cost and (if applicable) PV/load as features.

Linear regression and neural networks do not perform well on the test set. Random forests on the other hand achieve good results, with R^2>0.88; using additional regularization or feature cleaning allow to improve the performance only slightly (~2%). Support Vector Machines achieve the highest score, but the improvement is slight compared to RFs (1%).

It is worth to remember that for nTraces > 40 (approx.), the solution value is constant. This implies that the datasets (training and test sets) have several nTraces associated to the same features' values. Hence, a regressor that takes as feature cost, PV, and Load and predicts nTraces can not always predict the right value of nTraces. At test time the regressor will predict one value for nTraces; it might be not close to the ground truth but not necessarily wrong as the pattern itself in the datasets is ambiguous – in the training set, similar features are associated to several different values of nTraces from 40 to 100. The regressor can not reach a very high test error as the problem is ill-posed due to the data itself.

As noted before, for a number of traces lower than 40, the solution value changes when nTraces varies. In this situation, the instance also influences the solution value: as shown in the graphs in Appendix A, based on the instance there is a growing or decreasing trend in the cost-nTraces relationship. As a consequence, it is reasonable to use PV and Load as features because they are characteristic of each instance and thus they might help to capture relevant patterns.

ii. nTraces and time

Focus on the relationship between number of traces and computation time. Each model predicts nTraces using as feature only the time. The dataset suggests a strong polynomial relationship between the two variables, hence it is reasonable to model it with a linear regression with polynomial basis.

We use a linear regressor, and we explore different polynomial bases:

- Linear regression with change of basis to quadratic base, i.e. polynomial base with degree 2.
- Linear regression with change of basis to polynomial base with degree 3.
- Linear regression with change of basis to logarithmic base.

Basis	Pol 2	Pol 3	Log
R^2	0.98718	0.99573	0.99273

Table 2: Test set performance for regressors that predict nTraces using time as feature.

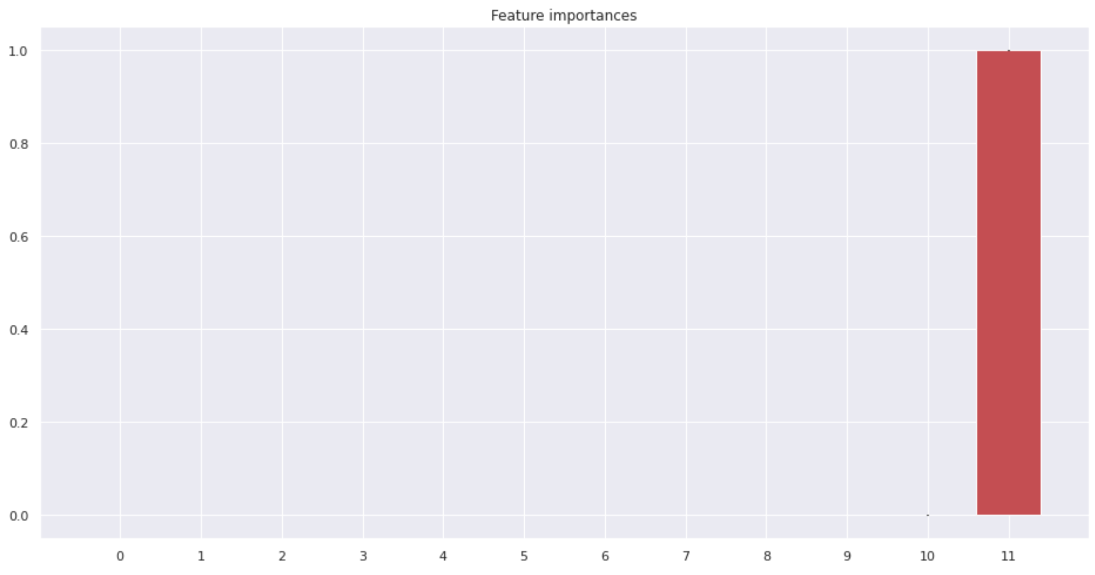
The models yield good performance on the test set, in accordance to the relationship found during the dataset analysis.

iii. nTraces and time/cost

We try to condense the previous two models into one, i.e. a unified model that captures the relationship between number of traces, solution cost and computation time. This model predicts nTraces taking as features time, cost, PV, and load.

The models are:

- Random forest classifier for performing dimensionality reduction of PV and Load (max depth 6, number of trees 5), followed by Extra Tree that performs the actual regression taking time, cost, and reduced PV and Load. The score is remarkably high. However, according to the feature importance value, the only feature that influences these predictions is time whereas the other variables have little importance, as shown in Graph 9.



Graph 9*:* Feature importance for the RF. Features 0-4 are PV, 5-9 are Load, 10 is cost, and 11 is time.

- SVM regressor with Radial Basis Function kernel and regularization parameter C=100.

Model	**RF**	**SVM**
R^2	0.9999	0.0

Table 3: Test set performance for regressors that predict nTraces using time, cost and (if applicable) PV/load as features.

According to the results, when we fix time the number of traces is determined and vice versa. This is consistent with the results in the previous sections: these two variables have a strong correlation, and this is captured by the unified model proposed here. As a consequence, if we specify the same time and different solution costs this model will predict the same value for nTraces. It is not possible to obtain different costs for the same time: this model captures the relationship between time and nTraces and not the relationship between the other variables, because the latter are weaker compared to the first.

iv. nTraces and memory

Focus on the relationship between online memory usage (either average or maximum) and computation time.

In the dataset analysis we explored the relationship between these variables. This information can be leveraged to guide the construction of empirical models:

- Random forest or SVM regressor to predict nTraces from average memory and instance information (PV and Load). PV and Load are relevant when nTraces has a large value, hence they should be used as features.
- Linear regression should capture the strong relationship between maximum memory and nTraces.

Best models, for each ML technique, that predict the number of traces given the average memory; they all take memory, PV, and Load as features:

- Lasso Linear regressor, i.e. Linear regressor with LASSO (least absolute shrinkage and selection operator) or L1 regularization.
- RF regressor (max depth 20), with RF Classifier (max depth 6, 5 DTs) for dim reduction of inputs. As shown by the feature importance (reported in Appendix B), the prediction is completely based on memory and it ignores PV and Load.
- SVM regressor with linear kernel.

Model	**LLR**	**RF**	**SVM**
R^2	0.65790	0.94722	0.76906

Table 4: Test set performance for the best regressors that predict nTraces using average memory and (if applicable) PV/load as features.

Models that predict the number of traces given the maximum Memory; they do not use PV and Load as features:

- Lasso regression with polynomial base with degree 3.
- SVM regressor with linear kernel.

Model	**LLR**	**SVM**
R^2	0.99777	0.98632

Table 5: Test set performance for regressors that predict nTraces using maximum memory as feature.

Random forests are the best predictors for average memory given nTraces when PV and Load are used as features, with high R^2 scores on the test set. On the other hand, linear regression and SVM do not capture the relationship well.

The relationship between nTraces and maximum memory is approximated well by a linear regression with polynomial basis.

v. Total

We study a complete model that predicts nTraces given all or some values of average memory, time, PV, Load. There are two possible design approaches for this system:

- Unified model. A unique regressor that takes as features all possible inputs.
  - Pros: a unique model is easier to handle compared to multiple models as it requires only one model design and training, hence it can be easily applied to different problems.
  - Cons: a unique regressor is less accurate because it must capture the relationships between all variables. This is not trivial, as showed in the previous section when a single model dealt with time and cost. Furthermore, it is not modular, i.e. adding or removing an input variable requires a new training of the complete model; this makes it less suitable for a flexible and configurable system.
- Separate models. Several regressors, each predicts nTraces for one of the input variables and is designed specifically on the relationship between its input and nTraces. The prediction of the single regressors are then aggregated to obtain a unique value.
  - Pros: it is more accurate as each regressor captures the specific relationship between nTraces and its feature. It is also modular, i.e. it is flexible and allows to remove or add input variables just by removing or adding single models rather than training a complete new unique model.
  - Cons: it is less straightforward to transport separate models to completely new problems, as they require a specific model design for each input variable.

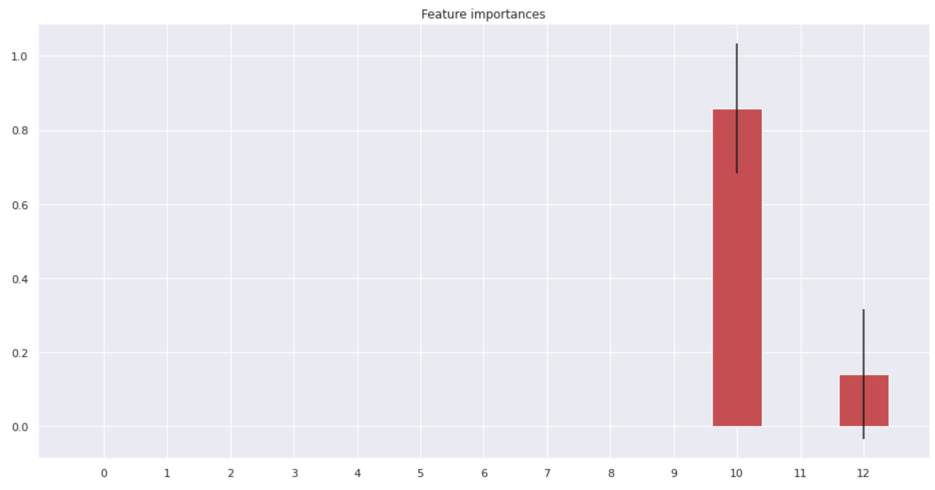
Unified models:

- PCA (5 principal components) for dimensionality reduction of PV and Load. Then, Extra Trees regressor with max depth 20.
- PCA (5 principal components) for dimensionality reduction of PV and Load. Then, NN with architecture [280], activation tanh, maximum epochs 500, using average memory. Similarly to the RFs, also in NNs time and memory have a major impact on nTraces prediction.
- SVM regressor with linear kernel.

Model	**RF**	**NN**	**SVM**
R^2	0.99999	0.98555	0.94796

Table 6: Test set performance for the best regressors that predict nTraces using all the remaining variables as features.

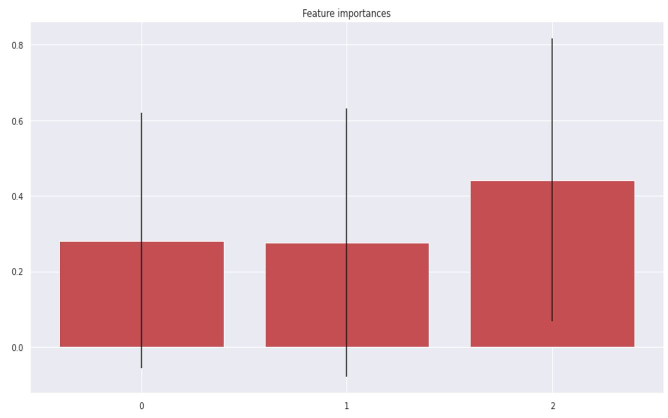
The models have high performance on test sets. However, they might not have the desired behavior. They predict nTraces by considering mostly time (the most important input) and a bit of memory, without taking into consideration other variables. This can be observed from the feature importance in random forests, reported in Graph 10. In other words, if we feed the model values for time, cost and memory, the model will predict nTraces bases on time and neglecting other features.



Graph 10: Feature importance for the RF. Features 0-4 are PV, 5-9 are Load, 10 is time, 11 is cost, and 12 is memory.

Separate models: we use separate regressors, one for each feature time, memory and cost, to predict nTraces. Each regressor is customized on the specific relationship between its input and the target, both in terms of model and of features (namely, whether to use PV and Load). Each regressor provides a suggested value for nTraces, and all the suggestions are aggregated to produce a single value.

- Single regressors:
    - Time. Linear regressor with logarithmic basis, only time as features: `0.99208`
    - Cost. PCA (5 components) for PV and Load followed by an Extra Trees regressor (depth max 40): `0.88972`
    - Average mem. PCA (5 components) for PV and Load followed by an Extra Trees regressor (depth max 40): `0.89107`
- Aggregate prediction. We explore different ways to aggregate the three values obtained with the regressors:
    - Minimum value suggested. This represents a conservative approach, as choosing the minimum nTraces implies using the minimum run-time and memory in the online phase.
    - Random forest with 10 trees, max depth 5.



Graph 11*:* Feature importance for the RF model that aggegates scores of single regressors. In order: score of the regressors for average memory, cost, and time.

    - Linear regression.
    - SVM regression with RBF basis.

Aggregation	Min	RF	LR	SVM
R^2	0.90401	0.97259	0.89382	0.57057

Table 7: Test set performance for the best regressors that predict nTraces using all the remaining variables as features.

Results show that separate regressors aggregate yield a good performance on the test set. Random forest is the best aggregation method, leading to an R^2 of 0.97.

5.3.1.2 Decision Trees

In this section we specifically focus on decision trees, since they are the most important models involved in the proposed system for the next experiments. We explore how DTs with different hyperparameters capture the relationships between variables.

For each setting (i.e. hyperparameters and dataset's normalization) we build four separate models, one model for each possible prediction target. Each model uses as features the three remaining variables. Average memory is considered rather than maximum memory because it is a better estimator of the resource required throughout the entire optimization process. The four models are:

- Regressor predicting computation time given memory, cost, and nTraces;
- Regressor predicting average memory given time, cost, and nTraces;
- Regressor predicting solution cost given time, memory, and nTraces;
- Regressor predicting number of traces given time, memory, and cost.

We compare models that adopt or do not adopt instance information (PV and Load) as features. PV and Load are used in an aggregate form: instead of taking the entire vectors, we use as features the vectors' mean values.

Hyperparameter tuning is not performed on decision trees in these experiments. For some DTs we specify a maximum depth, in order to examine the performance of shallow models. For other trees no maximum depth is specified in the training process: the DT is grown to the maximum depth possible.

We compare the performance on the original dataset and on its standardized and min-max scaled version. Standardization and min-max scaling are two different forms of data normalization:

- *Standardization*, sometimes called *z-score*: feature j for the sample i ($x_{i,j}$) is transformed to:

$$s_{i,j} = \frac{x_{i,j} - \mu_j}{\sigma_j} \tag{52}$$

  where μ_j is the mean and σ_j is the standard deviation for feature j over the training set. This process brings the transformed feature's distribution to have mean 0 and standard deviation 1. The test set is processed using μ and σ values from the training set.

- *Min-max feature scaling*: feature j for the sample i ($x_{i,j}$) is transformed to:

$$n_{i,j} = \frac{x_{i,j} - m_j}{M_j - m_j} \tag{53}$$

  where m_j is the minimum and M_j is the maximum value for feature j over the training set. This process brings all values of the transformed feature to the range [0, 1]. The test set is processed using m and M values from the training set.

Hyperparameters and test set performance for the different DT models are in Table 8. We report here for brevity only the standardized dataset because it is used in the final proposed model. Results for all dataset forms are in Appendix B.

Features	Target	DT depth bound	DT depth	R^2
No PV/Load as features	**nTraces**	1	1	0.7525
		No	7	1.0000
	Cost	5	5	0.0979
		No	54	-0.4925
	Time	No	22	1.0000
	Average Memory	1	1	0.5648

		No	23	0.9980
PV/Load as features	**nTraces**	1	1	0.7525
		No	7	1.0000
	Cost	5	5	0. 5219
		No	21	0.9431
	Time	No	17	1.0000
	Average Memory	1	1	0.5648
		No	22	0. 9914

Table 8: Test set performance of experimental DTs on the standardized dataset.

According to the results, the performance of decision trees is not heavily influenced by the normalization (i.e. standardization or min-max scaling) of data.

Prediction of nTraces, time, and average memory is not influenced by the usage of instance-specific data as features, i.e. PV and Load. On the other hand, the cost regressor experiences a significant decrease in test error when PV and load are added among features.

5.3.2 Final Empirical Models

We design and train the final models that will be embedded into the combinatorial optimization problem with EML: decision trees and neural networks.

These two ML techniques were studied and proved suitable in literature for modeling the runtime behavior of optimization algorithms [19]. This motivates their adoption in the model proposed in this work. Moreover, they are the main empirical models supported by EML. We do not select random forests as they are supported by EML but not extensively tested. Furthermore, they do not lead to significant improvements in terms of performance on the test set compared to decision trees. Hence, we focus our study of non-GD-based models on DTs.

For each setting (ML technique and dataset's normalization) we build 3 models, one for each target variable among solution cost, average memory and resolution time.

They are slightly different from the ones introduced in the previous section: the target is predicted based on nTraces and (if applicable) PV/Load. In other words, regressors only take nTraces and PV/Load as features and they do not use other variables. As mentioned before, we adopt average memory because it is a better estimator of the resources used throughout the entire optimization process, compared to maximum memory.

5-fold Cross-validation is used to perform hyperparameter tuning, with the aim of selecting the tree's depth or the NN's architecture and training setting.

We experiment with the three different data normalization forms: original, standardized and min-max scaled. We build several DT and NN models:

- Decision trees with maximum depth max 5, 9, 11 or unbound. This is not necessarily the real depth of the trees; it is the maximum depth, after which the tree expansion is stopped during training. In this setting, cross-validation chooses the best depth from 1 to the max depth specified.
- Neural networks. Cross-validation is used to select:
    - The architecture among three choices, where each layer has 100 neurons and the number of layers varies from 2 to 4. Namely, [100, 100], [100, 100, 100], and [100, 100, 100, 100].
    - The activation between ReLU and tanh.
    - The batch size among 100, 200, and 400.
    - The number of epochs among 50, 100, and 500.

5.3.2.1 Training Results

For each model we report hyperparameters selected with cross-validation, R^2 score on the test set and training time. It is important to distinguish between two times: *resolution time* is a variable in the dataset and it is the time taken by the online optimization phase; *training time* (*t.time*) is the time used for training the ML model. In blue results for the models adopted in our system, as explained in the next section.

Times are in seconds. For data normalization forms, *Orig.* is the original dataset, *Std.* is the standardized version and *M.m.s.* is the data with min-max feature scaling.

DT_5 - DT maximum depth 5:

Features	**Data Norm.**	**Model**								
		Solution Cost			**Resolution Time**			**Average Memory**		
		Depth	**R^2**	**T.Time**	**Depth**	**R^2**	**T.Time**	**Depth**	**R^2**	**T.Time**
nTraces	**Orig.**	3	0.0866	1.2096	5	0.9992	1.2812	5	0.8342	1.2651
	Std.	3	0.0866	1.2438	**5**	**0.9992**	**1.2896**	**5**	**0.8342**	**1.2805**
	M.m.s.	3	0.0866	1.2762	5	0.9992	1.2538	5	0.8342	1.3185
nTraces, PV and Load	**Orig.**	5	0.5231	2.3500	5	0.9992	2.4220	5	0.8659	2.3673
	Std.	**5**	**0.5232**	**2.3023**	5	0.9992	2.3598	5	0.8659	2.3750
	M.m.s.	5	0.5231	2.2692	5	0.9992	2.3097	5	0.8659	2.3485

Table 9: Test set performance, depth, and training time of DT$_5$.

DT_9 - DT maximum depth 9:

Features	**Data Norm.**	**Model**								
		Solution Cost			**Resolution Time**			**Average Memory**		
		Depth	**R^2**	**T.Time**	**Depth**	**R^2**	**T.Time**	**Depth**	**R^2**	**T.Time**
nTraces	**Orig.**	3	0.0866	2.4186	9	0.9999	2.3729	9	0.9019	2.4110
	Std.	3	0.0880	2.5610	**9**	**0.9999**	**2.4757**	**9**	**0.9019**	**2.4831**
	M.m.s.	3	0.0880	2.3363	9	0.9999	2.5101	9	0.9019	2.5122
nTraces, PV and Load	**Orig.**	9	0.8645	9.6251	9	0.9999	4.4393	9	0.9566	4.5104
	Std.	**9**	**0.8685**	**4.4050**	9	0.9999	7.4907	9	0.9566	6.4911
	M.m.s.	9	0.8645	4.3009	9	0.9999	4.3908	9	0.9556	4.4500

Table 10: Test set performance, depth, and training time of DT$_9$.

DT_{11} - DT maximum depth 11:

Features	**Data Norm.**	**Model**								
		Solution Cost			**Resolution Time**			**Average Memory**		
		Depth	**R^2**	**T.Time**	**Depth**	**R^2**	**T.Time**	**Depth**	**R^2**	**T.Time**
nTraces	**Orig.**	3	0.0866	2.8448	9	0.9999	2.2116	11	0.9032	2.6202
	Std.	3	0.0866	2.6708	**9**	**0.9999**	**2.2992**	**11**	**0.9032**	**2.6954**
	M.m.s.	3	0.0866	2.6737	9	0.9999	2.1524	11	0.9032	2.6644
nTraces,	**Orig.**	11	0.9338	10.0952	10	0.9999	5.9117	11	0.9637	5.0474

PV and Load	**Std.**	**11**	**0.9391**	**4.9230**	10	0.9999	5.0698	11	0.9619	7.5382
	M.m.s.	11	0.9342	4.9433	10	0.9999	5.0030	11	0.9662	7.3752

Table 11: Test set performance, depth, and training time of DT_{11}.

DT₁₅ - DT maximum depth unbound:

Features	**Data Norm.**	**Model**								
		Solution Cost			**Resolution Time**			**Average Memory**		
		Depth	**R^2**	**T.Time**	**Depth**	**R^2**	**T.Time**	**Depth**	**R^2**	**T.Time**
nTraces	**Orig.**	3	0.0866	3.2392	8	0.9999	1.8410	10	0.9029	2.2193
	Std.	3	0.0866	3.2422	**8**	**0.9999**	**1.8698**	**10**	**0.9029**	**2.2474**
	M.m.s.	3	0.0866	3.3010	8	0.9999	1.8588	10	0.9029	2.0783
nTraces, PV and Load	**Orig.**	18	0.9465	7.6124	10	0.9999	9.9832	14	0.9640	9.7557
	Std.	**15**	**0.9546**	**7.5718**	10	0.9999	14.4162	20	0.9715	8.9079
	M.m.s.	17	0.9489	7.5641	10	0.9999	11.8649	20	0.9684	8.9700

Table 12: Test set performance, depth, and training time of DT_{15}.

NN - NN; hyperparameters selected for each model are reported in Appendix C:

Features	**Data Norm.**	**Model**					
		Solution Cost		**Resolution Time**		**Average Memory**	
		R^2	**T.Time**	**R^2**	**Time**	**R^2**	**T.Time**
nTraces	**Original**	0.0845	713.4111	0.9999	729.8035	0.6694	740.0153
	Std.	0.0928	717.4230	**0.9999**	**733.1245**	**0.7896**	**726.6838**
	M.m.s.	0.0824	729.0209	0.9999	726.8239	0.7411	735.1767
nTraces, PV and Load	**Original**	0.1272	734.1813	0.9998	726.3660	0.7178	740.6667
	Std.	**0.9295**	**733.1343**	0.9999	725.0056	0.8050	727.2530
	M.m.s.	0.9210	741.6749	0.9999	728.7338	0.8017	729.1535

Table 13: Test set performance and training time of NNs.

Neural networks have significantly higher training times compared to any decision tree. Among DTs, larger depth is related to a slight increase in training time.

Performance on the test set for DTs gets better when the depth is larger. DT_{15} achieves the best errors while in DT_{11} and DT_9 it is slightly larger. There is a significant gap in performance between these models and DT_5, in particular in the prediction of cost (respectively, R^2 of 0.95+ and 0.52), most likely indicating a severe overfitting of the latter compared to other trees. The largest difference in performance is for the cost regressor, while prediction of time is not heavily influenced by hyperparameters and it is very high in all cases, with R^2 >0.99. Neural networks achieve similar performances in the prediction of cost and time, but lead to a lower test score on memory compared to the worst DT model: 0.79 for NNs and 0.83 for DT_5.

Remarkably, the performance of neural networks is better in terms of test scores when the datasets are normalized, namely, standardized or min-max scaled. Normalization of datasets has a beneficial effect in the objective function of the neural network [39] [40] [41], leading to faster and more stable convergence during gradient descent; this explains a better test score in that situations. On the other hand, the performance of decision trees is not influenced by normalization on the training set. The better NN performance motivates the adoption of standardized data in the final models.

For both NNs and DTs, models predicting time and average memory do not perform significantly better when information of the instance (PV and Load) is used among the features. On the other hand, predictors of the solution cost that take into account PV and Load outperform models that do not use instance-specific features. We decide to use only nTraces as feature for both memory and time regressors, obtaining similar predictions with less complex ML models; on the other hand, we use nTraces, PV, and Load for models predicting the cost.

5.3.2.2 Final models

We select some of the machine learning models proposed in section 5.3.2.1, with the guidance of the results reported. These will be embedded in the high-level optimization problem with EML. In particular:

- We compare each technique and hyperparameter setting mentioned in the previous section. Namely, we adopt the following groups of models:
    - *DT_5*: decision tree with maximum depth in training 5.
    - *DT_9*: decision tree with maximum depth in training 9.

  - *DT_{11}*: decision tree with maximum depth in training 11.
  - *DT_{15}*: decision tree with maximum depth in training unbound. The deepest tree selected with this setting has depth 15, hence the name.
  - *NN*: neural network.
- For each technique we use three separate regressors, one for each target variable:
  - Solution cost based on nTraces, PV, and Load;
  - Average memory based on nTraces;
  - Solution time based on nTraces.

  The features for each regressors are selected in order to obtain the simplest model possible yielding satisfactory test results; details on this selection are in the previous section.
- We use the standardized datasets, as NN lead to a better performance on these data compared to their original form.

Complete details on models' architecture, training time, and test performance are reported in the previous part of section 5.3.2.

5.4 Combinatorial Optimization Model

In this section we provide information on the combinatorial optimization model that is proposed and tested in our experiments. This model represents the high-level optimizer within the entire system.

5.4.1 Optimization Model

The combinatorial optimization model in the proposed system has the following formulation. (56) is the EML-specific contribution while (54), (55), and (57) are the core combinatorial structure:

$$min\, f(x, z) \quad (54)$$

$$g_j(x, z), \forall j \in J \quad (55)$$

$$z = h(x) \quad (56)$$

$$x_i \in D_i \ , \forall\, x_i \in x \quad (57)$$

Variables specifically involved in the model are:

- *nTraces*: number of traces;
- *time*: computation time for the online optimization phase;
- *avgMem*: average memory used in the online optimization;
- *cost*: solution cost obtained by the hybrid online/offline optimizer;
- *PV* and *Load*: instance-specific data; they are the aggregated versions of the sequence of PV and Load values, namely, their average.

Because the high-level optimizer incorporates the behavior of both the controlled system (VPP) and the low-level optimizer (hybrid online/offline algorithm), variables carry information about both these systems.

The optimization model contains, among g_j in (55), some constraints that force nTraces to be an integer. Although the number of traces is an integer, nTraces is treated as a numerical value by the ML models. A possible solution to obtain correct nTraces values is to keep the variable as numerical throughout the entire optimization process, and round its value to an integer in the final solution. However, with this approach the optimization process is not aware of the real constraints; this might result in solutions that are not actually feasible. Therefore, we decide to enforce the type for nTraces already in the optimization model as additional constraints.

The ML models embedded in the optimization problem are the 3 regressors specified in section 5.3.2. Their encodings are represented by equation (56).

The core combinatorial structure (54), (55), and (57) depends on the specific problem. The optimization problem is flexible and the system is interactive: constraints and the objective function are specified by the user. In other words, the user specifies the core combinatorial structure desired for the use case.

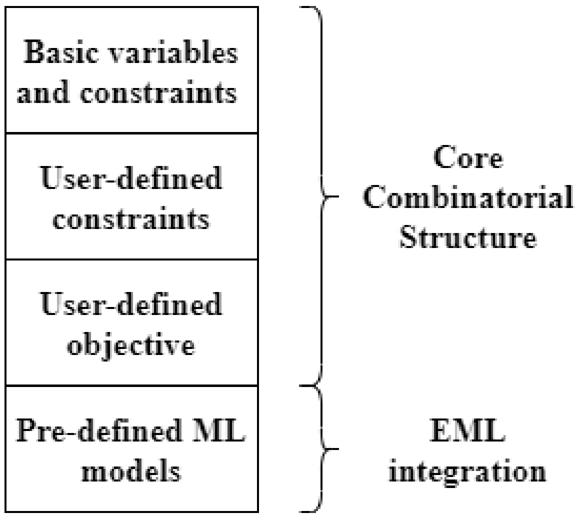


Figure 15: Composition of the custom combinatorial optimization model. It builds over a set of basic variables and constraints. Empirical ML models are integrated via EML. The objective and additional constraints are interactively specified by the user to fit the specific use case.

The modeling process of the optimization problem does not require domain-specific knowledge on the VPP or on the low-level optimizer. They can be treated as a black box, whose input/output relationship is modeled through machine learning and incorporated leveraging EML.

The combinatorial optimization problem is a prescriptive model, but it is data-driven thanks to the integration of ML provided by EML. The system is customizable as it allows the definition of the desired constraints and objectives. It is also highly flexible: it can be easily used for either deciding the low-level optimizer's configuration or forecasting its behavior, namely, the problem's input and output (constraints and optimization objective) can be easily specified based on the user's needs.

5.4.2 Optimization Approach

The optimization problem in our system is a MILP (mixed-integer linear programming) [42], modeled and solved using the IBM cplex solver. EMLLib[10] provides functionalities to embed machine learning models into this solver.

Mixed-integer linear programming (*MILP*) is a field of mathematical programming that addresses linear problems with continuous and integer decision variables. It provides techniques to find extreme points of linear objective functions with linear or integrality constraints. MILP solvers can leverage the problem's structure (i.e. constraints and cost function) to improve the efficiency of the search process, by adopting techniques such as linearization, cutting planes, branching, and constraint propagation.

5.4.3 Empirical Model Learning

In previous sections we detailed the machine learning models used to approximate the input-output relationship in the complex optimization system. Now, we embed the trained empirical model into the high-level combinatorial optimization problem. We adopt EMLLib[10], library that implements EML and is associated to [34]. Once the Empirical ML model has been encoded, its equations are automatically taken it into account by the solver for boosting the search process, e.g. for computing bounds and generating cuts.

In order to avoid re-building the optimization model for every experiment, we construct and save a unique base model containing:

- Basic variables and constraints;
- Embedded ML models.

[10] *https://github.com/emlopt/emllib*

All experiments start from this model and add custom constraints and objectives based on the specific requirements.

Below we report information on the embedded models. Times reported are the sum for all three models. The embedding time is the time needed to embed the three empirical models into the optimization problem with EML. The loading time is the time needed to load the pre-constructed base model from disk.

Model	ML model info	Combinatorial optimization model info				
	Training time (sec)	Embedding time (sec)	Loading time (sec)	Variables number	Constraints number	Model size on memory
DT_5	4.8725	0.1201	0.0335	100	457	104 KB
DT_9	9.3638	7.2999	0.8926	571	4596	7.77 MB
DT_{11}	9.9177	48.3547	4.1838	1064	10295	46 MB
DT_{15}	11.6890	466.6453	27.2517	2119	25455	1.03 GB
NN	2192.9426	2.9675	0.2445	2854	1003	1.06 MB

Table 14: Information on times and dimensions of combinatorial optimization models with embedded ML models.

According to the results, ANNs are significantly slower in training compared to DTs. Among DTs, training is faster when the DT has a smaller depth.

Remarkably, all trees have less variables compared to the ANN but a significantly larger amount of constraints, except for DT_5. This results in DTs with depth 9, 11 and 15 to have slower embedding/loading times and larger size with respect to ANNs, while this is not true for depth 5 trees. These trends are consistent with the models' architecture and with how EML represent them. The number of neurons in the NN is relatively large, and the number of variables introduced by EML has a roughly linear correlation with the number of neurons in a NN and with the depth of a DT. The number of paths in a tree grows significantly when the tree gets deeper and broader, and so does the amount of constraints used by EML to encode it.

In DTs the model's depth heavily influences the optimization model. Deep trees have a greater number of both constraints and variables compared to smaller models. The size that the optimization model takes on disk depends on these factors: it is larger when the trees are deeper, reaching a very large 1.03 GB size for DT_{15}. The times to embed trees and to load the pre-formed optimization model are related to the amount of variables and constraints. As a consequence, they increase significantly when the depth increases; for example, the embedding time and loading time are respectively 60 and 27 times larger in DT_9 compared to DT_5.

Chapter 6
Experiments and Results

Experiments are focused on the behavior of the high-level optimizer, inserted on top of the low-level optimizer. The latter is the hybrid offline/optimization technique that leverages the fixing heuristic and a contingency table.

We perform two groups of experiments. First, we examine the performance of optimization models when the embedded empirical models are either decision trees or neural networks. We test them on several comparative experiments aimed at finding weaknesses and strengths of models. Then, guided by the results of the preliminary experiments, we select the best machine learning models to use in the high-level optimizer, and we test the resulting system in two real-world scenarios.

The empirical models used in the following experiments are the ones detailed in section 5.3.2, and the combinatorial optimization model is detailed in section 5.4.

6.1 Comparative Experiments

In comparative experiments on decision trees and neural networks each model under test is applied to the same problem in a similar setting. We examine the performance of the two ML techniques by means of time to solve the optimization problem and solution quality. This allows us to select the model to use in the complete experiments reported in the next section.

We compare different DTs hyperparameters, namely, different maximum depths used for training: 9, 11, and unbound. Information the hyperparameters of both DTs and NNs are reported in section 5.3.2.

Several problems are considered in these experiments:

- Minimize the number of traces, given constraints on the solution value.

- Minimize the solution value, given constraints on memory or time.
- Minimize the time, given constraints on memory or solution cost.
- Minimize the memory, given constraints on solution cost.

We perform these experiments for different values of the variables involved in constraints. Experiments are designed to compare and find weaknesses in models. We explore both normal values and edge cases, i.e. constraints' values very close to the domain limit for the variable, in order to shed light on differences in behaviors among models.

The optimization models in these experiments are not instance specific: we do not constrain PV and Load to particular values.

6.1.1 Results

Below are the solutions found for the optimization problem and the resolution times. We report here, for brevity, only relevant problems; other experiments highlight similar behaviors and are described in Appendix D.

Variables involved in the solution change based on the problem, thus they are reported one per row. Times are in sec. A timeout of 1200 sec is set for the solver: after that time, the resolution process is stopped and the solution returned by the solver is the current optimum point, if existing. It is important to distinguish between two times: *time* is a variable in the dataset and it is the time taken by the online optimization phase; *resolution time (Res. time)* is the time taken by our high-level optimizer to find a solution.

Objective (minimized)	**Constraints**	**Solution Variable**	**Empirical ML Model**							
			DT_9		**DT_{11}**		**DT_{15}**		**NN**	
			Value	**Res. time**	**Value**	**Res. time**	**Value**	**Res. time**	**Value**	**Res. time**
nTraces	Cost <= 398	nTraces	1	0.2191	1	1.5095	1	5.1944	No solution	1200
		Mem	88.92		88.92		88.92			

		Time	6.93		6.93		9.51			
	Cost <= 256	nTraces	No solution	0.1494	24	1.2049	24	6.2743	No solution	1200
		Mem			2953.95		2953.95			
		Time			381.99		381.99			
Cost	Mem <= 3963, Time <= 2128	Cost	255.91	0.2724	254.83	2.0288	254.83	6.6248	453.06	1200
		nTraces	13		24		24		37	
	Mem <= 819, Time <= 71	Cost	268.10	0.2346	268.10	1.1606	268.10	4.3203	337.06	1200
		nTraces	3		3		3		8	
Time	Cost <= 398	Time	6.93	0.2190	6.93	1.6775	9.51	5.3994	5.56	593.4029
		nTraces	1		1		1		2	
	Cost <= 256	Time	No solution	0.1834	381.99	1.2097	381.99	5.8430	No solution	1200
		nTraces			24		24			
Mem	Cost <= 256	Mem	No solution	0.1493	2953.95	1.1909	2953.95	5.8295	6166.68	1200
		nTraces			24		24		77	

Table 15: Comparative experiments for DT_9, DT_{11}, DT_{15}, and NN. For each problem we report objective and constraints, resolution time for the high-level optimizer and, for each variable of interest, the solution value.

According to the results, optimization models derived from ANNs are significantly slower in obtaining solutions compared to all DTs-based models. Among the latter, the depth of the trees influences the optimization solving time: increasing the tree depth results in slower resolution time.

NNs in some problems do not find any solution in useful time (1200 sec); however, they find a solution if constraints for these problems have relaxed values. This is most likely a consequence of NNs not predicting the values requested by the constraints for those variables. When we constraint values at the boundaries of the variables' domains, DTs find reasonable solutions while NNs do not. This is a sign that NNs underfit compared to DTs.

From these experiments we can examine how DTs' depth influences the results of the optimization problem. DT_9 has faster embedding time and resolution time (0.20 - 0.40 sec), but struggles in finding solutions for extreme values, similarly to a neural network, and often solutions are not consistent with deeper trees. DT_{11} has slightly slower embedding and resolution (res time 0.90 - 1.60 sec); it finds all the solutions, they are almost always equal to the deep tree even if not as good as DT_{15} for extreme values.

These experiments are important as they demonstrate that the proposed optimization system not only minimizes the requested objective and respects constraints, but also gives suggestions on other variables.

6.2 Final Use Cases

The proposed system is tested on two case studies. They are similar to real-world scenarios where the optimizer could be employed:

- *Optimal Number of Traces*: Find the low-level optimizer's configuration (i.e. number of traces) that yields the best cost given time and memory constraints. Given time/resources requirements specified by the user, namely, constraints limiting computation time or memory, minimize the solution cost. The optimizer outputs the number of traces required to obtain this solution value, together with the predicted solution cost, memory and time.
  In this problem:
    - Constraints: memory and resolution time.
    - Objective: minimize solution cost.
    - Output: number of traces; forecast on solution cost, memory, and time.
- *Solution Improvement over Baseline*: Find the optimal time/resources configuration (i.e. time and memory) required to reach a solution improvement with respect to the greedy heuristic.
  Each instance is associated to a baseline solution value, given by the cost obtained with the greedy heuristic (section 2.1). Given a value of improvement for the solution cost w.r.t. the baseline specified by the user, i.e. a constraint on the solution cost, minimize the number of traces to obtain it. The optimizer

outputs memory and time required to obtain the desired improvement, together with the predicted number of traces and solution cost.
In this problem:

- Constraints: solution cost.
- Objective: minimize number of traces.
- Output: memory and resolution time; forecast on number of traces and solution cost.

We experiment on these scenarios only with decision tree models, with different maximum depths in training (5, 9, 11, and 15); information on these DTs are in section 5.3.2. We do not experiment with neural networks given the poor performance on the test set; additionally, according to results in section 6.1, they yield optimization models that are significantly slow in the resolution process and, as a consequence, they can not find useful solutions in many cases.

The optimization problem is structured as specified in section 5.4.1. Constraints in the core combinatorial structure are the constraints for the specific experiment, defined by the user. We also add two constraints that bind the PV and Load variables in the optimization problem to the values of the instance considered in the specific run; this allows the optimization problem and its solution to be instance-specific.

We test on 42 new and unseen instances; they were not present in the dataset used to build ML models. Each instance was solved with the greedy heuristic, yielding a solution cost that represents a baseline for the hybrid offline/optimization approach.

For the first use case (Optimal Number of Traces):

- We experiment on a set of 5 values for average memory and a set of 7 values for computation time. We perform the experiments adopting a grid-search-like approach, i.e. we fix one value for the memory and perform experiments for each time value, with one run for each instance and for each combination of values.

  - The average memory values are (in MB): 512, 1024, 2048, 4096, 8192
  - The solution time values are (in sec): 50, 70, 90, 110, 250, 500, 750

For the second use case (Solution Improvement over Baseline):

- We experiment on a set of values for solution cost improvement w.r.t. the baseline. We perform one run for each instance.
  - The improvements in solution cost over the baseline (in %) start from 2% and sequentially increase by 2% or 3% until no solution is found, i.e.: 2, 5, 7, 10, 12, 15, 17…

6.2.1 Results

We report in this section solutions obtained by the high-level optimizer on the two use cases under examination, together with information on resolution times.

6.2.1.1 Optimal Number of Traces

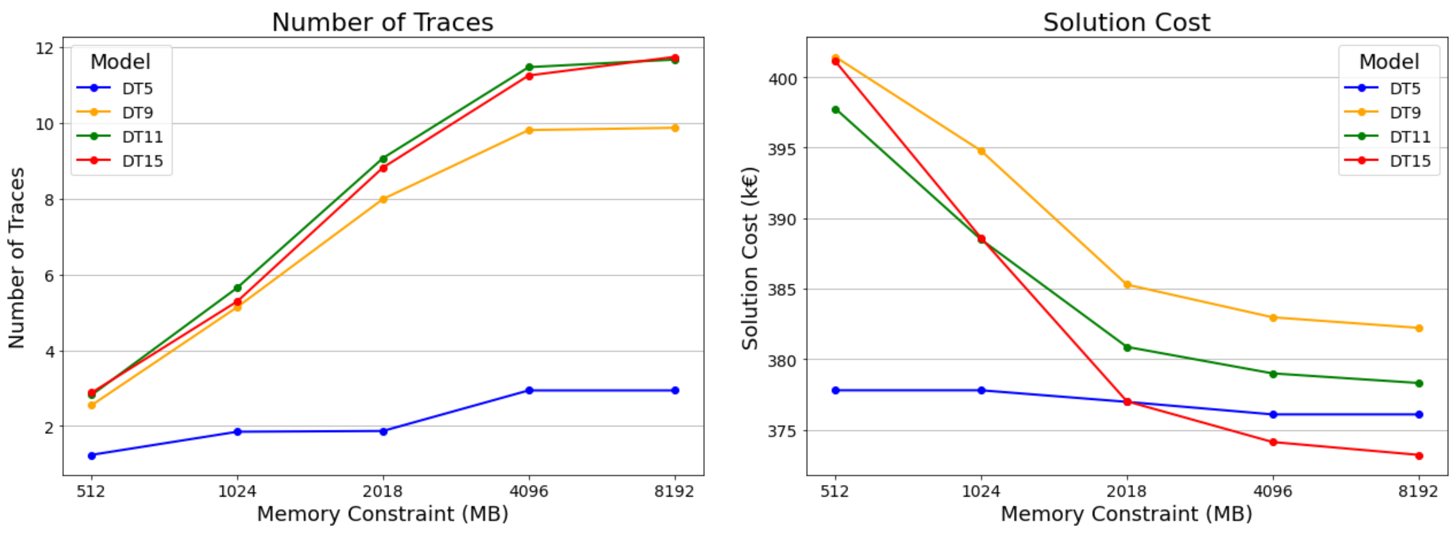
Table 16 documents the average resolution times for the optimization problem for all ML models. This is the time taken by our high-level solver to find a solution; it is not to be confused with the variable *time*, that represents the resolution time of the low-level hybrid optimizer and is a variable in the dataset.

	Empirical ML Model			
	DT_5	**DT_9**	**DT_{11}**	**DT_{15}**
Res. Time (sec)	0.0040	0.1073	0.6196	4.7341

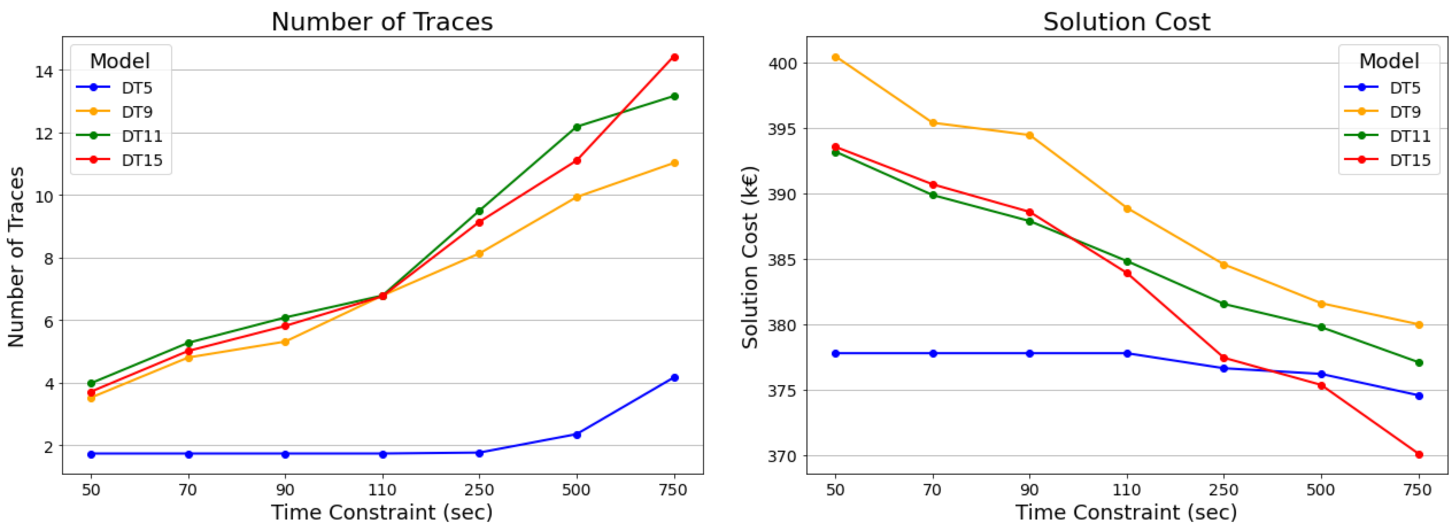
Table 16: Average resolution time for the high-level optimizer.

According to the results, the decision tree's depth heavily influences the resolution time of the proposed high-level optimizer. Increasing the depth from 5 to 15 leads to a growth in time by a factor of more than 10^3.

For each memory constraint value, we average the optimization results (i.e., nTraces suggested and solution cost forecasted) across all instances and time constraint values. We report in Graph 12 how these values change when the memory constraint value varies, for all the proposed ML models. The same operation is repeated with memory and time switched in Graph 13.



Graph 12: Number of traces and Cost suggested by the optimizers under different memory bounds. The proposed ML models are compared. For each memory costraint value, the result is averaged across all instances and time constraint values.



Graph 13: Number of traces and Cost suggested by the optimizers under different time bounds. The proposed ML models are compared. For each time costraint value, the result is averaged across all instances and memory constraint values.

Trends emerging from these results are not surprising. When we allow more memory and/or time (i.e. constraints are relaxed) the solution gets better in terms of cost (i.e. lower cost) compared to tighter bounds; at the same time, the optimal number of traces increases as more traces required to reach a better solution. Remarkably this trend is not captured by DT_5; for this model the solution cost is approximately constant for all time and memory values, while nTraces changes only slightly. DT_5's behavior is a typical evidence of underfitting, as it indicates that the model did not learn a relationship between the predictors and the target variables.

In general, results show that trees with depth 9,11, and 15 find similar solutions whereas DT_5 leads to results that are quite different compared to the other models.

For each constraint (i.e. fixing the values of memory and time constraints) we compute the standard deviation of the solution's values (i.e. cost and nTraces) found across all instances. We report here, for each model, this standard deviation averaged across all constraints' values:

Solution Variable	**Empirical ML Model**			
	DT_5	**DT_9**	**DT_{11}**	**DT_{15}**
Cost	37.21	56.70	57.36	55.04
nTraces	2.86	3.59	3.19	3.15

Table 17: Standard deviation of each solution value across all instances, averaged across all experiments (i.e. constraints' values) for each ML model.

Solutions in the shallowest DT (DT_5) are not significantly influenced by the instance, as demonstrated by a smaller standard deviation in results; feeding different instances (i.e., PV and Load values) to the same problem leads to the same solution. DT_5 is also less sensible to the value of constraints, i.e. the solution found by the optimizer is almost always the same despite the constraint. This behavior is not present in deeper trees and it is most likely due to underfitting of DT_5 compared to the others. It not a desirable behavior; in fact, we aim at obtaining a model that performs specific predictions based on the instance's characteristics.

Below are solutions found for the optimization problem based on DT_{11}, averaged across all instances, for some of the constraints value experimented:

Constraints		**Solution**			
Mem_c	**$Time_c$**	**nTraces**	**Mem**	**Time**	**Cost**
512	50	2.82	369.37	17.33	397.77
	110	2.82	369.37	17.33	397.77
	500	2.82	369.37	17.33	397.77
	750	2.82	369.37	17.33	397.77
1024	50	4.27	579.50	28.35	392.03
	110	5.88	755.93	44.20	387.89
	500	5.88	755.93	44.20	387.89
	750	5.88	755.93	44.20	387.89
2048	50	4.27	579.50	28.35	392.03
	110	8.39	1068.37	74.41	379.49
	500	12.82	1532.35	146.36	374.04
	750	12.09	1450.42	134.71	374.04
8192	50	4.27	579.50	28.35	392.03
	110	8.39	1068.37	74.41	379.49
	500	19.70	2455.50	307.24	369.61
	750	23.21	2869.46	432.03	360.55

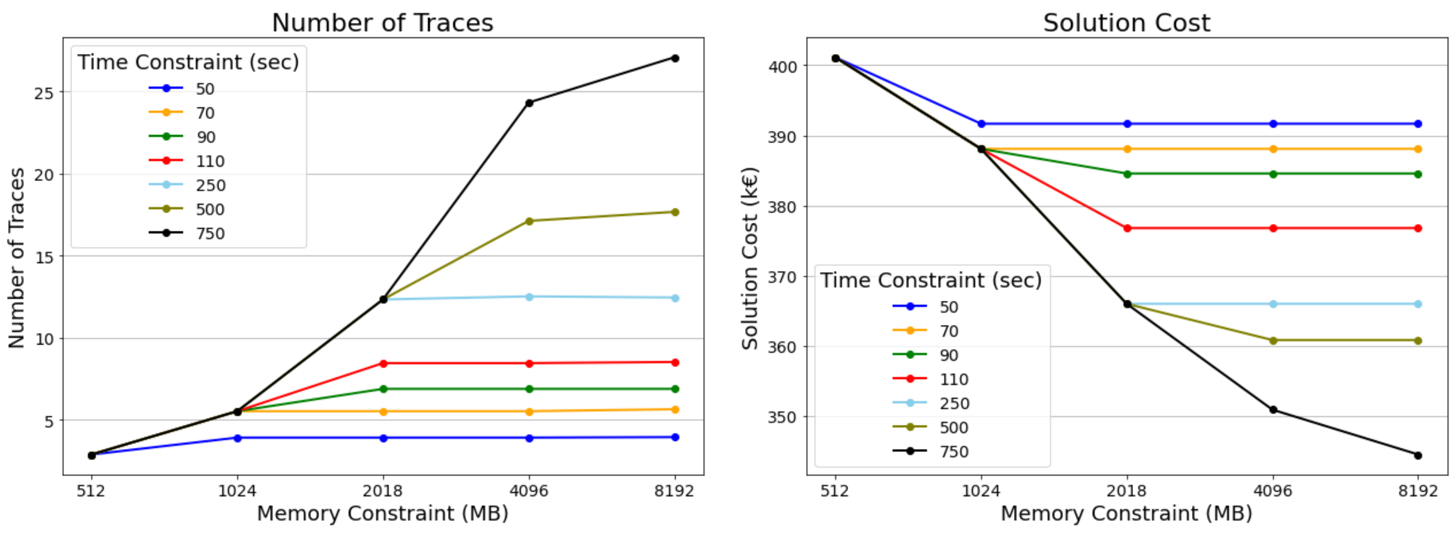
Table 18: Solutions of the high-level optimization model based on DT_{11}, averaged across all instances. The first two columns (mem_c and $time_c$) are the constraints. The remaining columns are the solution found: in this specific problem, nTraces are the variable suggested by the system whereas memory, time, and cost represent forecasts.

In general, these results demonstrate that the proposed system is able to suggest the optimal configuration (i.e. number of traces) given time/resources constraints, and at the same time to make predictions about cost, time, and memory.

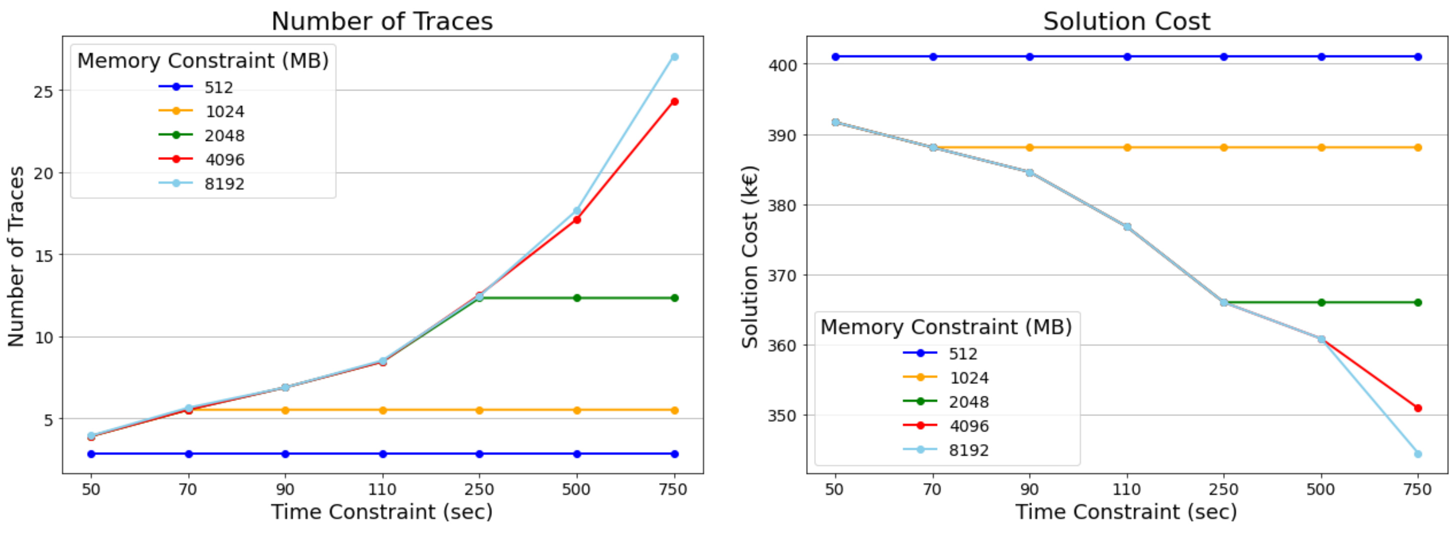
The optimizer finds feasible solutions, as demonstrated by the fact that time and memory in each solution are below the requested values. As expected, for tight memory constraints (i.e. small values of the memory bound), the solution memory value is closer to its bound compared to how the solution time is to the time bound.

Vice versa, for tight time bounds the result's time is closer to its constraint compared to memory.

The two graphs below give a complete and clearer vision of trends emerging from Table 18. For the same optimization problem, in Graph 14 we plot how results (nTraces suggested and solution cost forecasted) change when the memory constraint varies, for each value of the time constraint. Graph 15 reports the same operation with memory and time switched. All constraints' values experimented are reported, and results are averaged across all instances.



Graph 14: nTraces and Cost suggested by the DT_{11}-based optimizer under different memory bounds, averaged across all instances, for each time constraint value.



Graph 15: nTraces and Cost suggested by the DT_{11}-based optimizer under different time bounds, averaged across all instances, for each memory constraint value.

Remarkably, for low values of the memory bound (e.g. 512 MB) the optimization result is not affected by the time constraint imposed: increasing the time yields the same result, as we can see in Graph 15. On the other hand, when the memory constraint is less strict the result changes for different time values. For middle values of this constraint, the result changes for small values of time until a specific point, after which it remains constant when the time bound changes. Graph 14 show that this trend is also present when variables are switched; if the time constraint is tight (e.g. 50 sec), the best result is obtained with a low memory value (bound 1024 MB) and increasing the memory bound does not improve the solution.

This result is particularly interesting as it demonstrates that the high-level optimizer is able to suggest optimal configurations with consistent time/memory values for the low-level optimizer: if one of the constraints specified by the user is tight, our system suggests the configuration that yields the best solution value reachable with that constraint and that, at the same time, uses the smallest resources possible. For example, for a strict time bound, the suggested nTraces value is the configuration with lowest memory possible among all values that lead to the best solution cost; increasing the memory allowance does not change this solution.

6.2.1.2 Solution Improvement over Baseline

Below is the average resolution time for the proposed optimizer depending on the machine learning model.

	Empirical ML Model			
	DT_5	**DT_9**	**DT_{11}**	**DT_{15}**
Res. Time (sec)	0.0077	0.1183	0.5888	4.0262

Table 19: Average resolution time for the high-level optimizer.

Results are consistent with the ones found in the previous experiment: as the decision tree’s depth increases, the time to solve the combinatorial optimization problem grows.

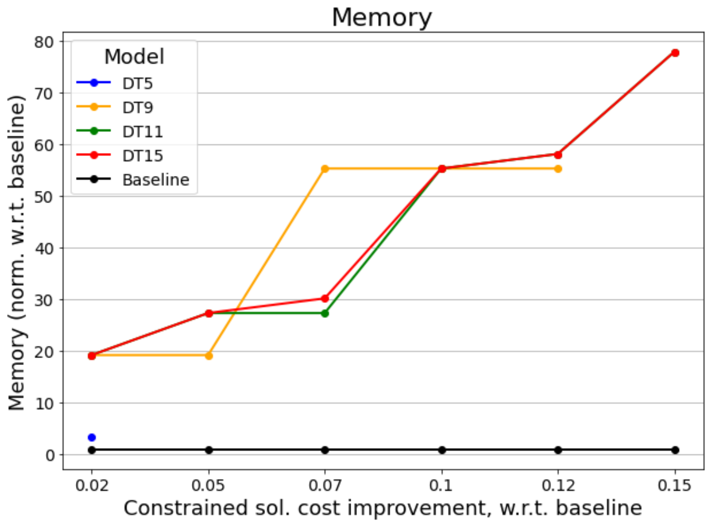
We report the maximum cost improvement found by each optimization model, averaged across instances and all experiments:

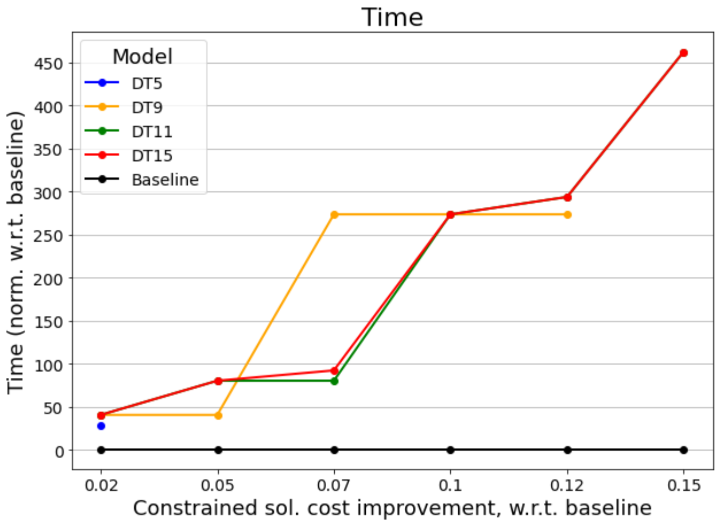
	Empirical ML Model			
	DT$_5$	**DT$_9$**	**DT$_{11}$**	**DT$_{15}$**
Improvement w.r.t. baseline	6.09%	10.13%	10.60%	12.65%

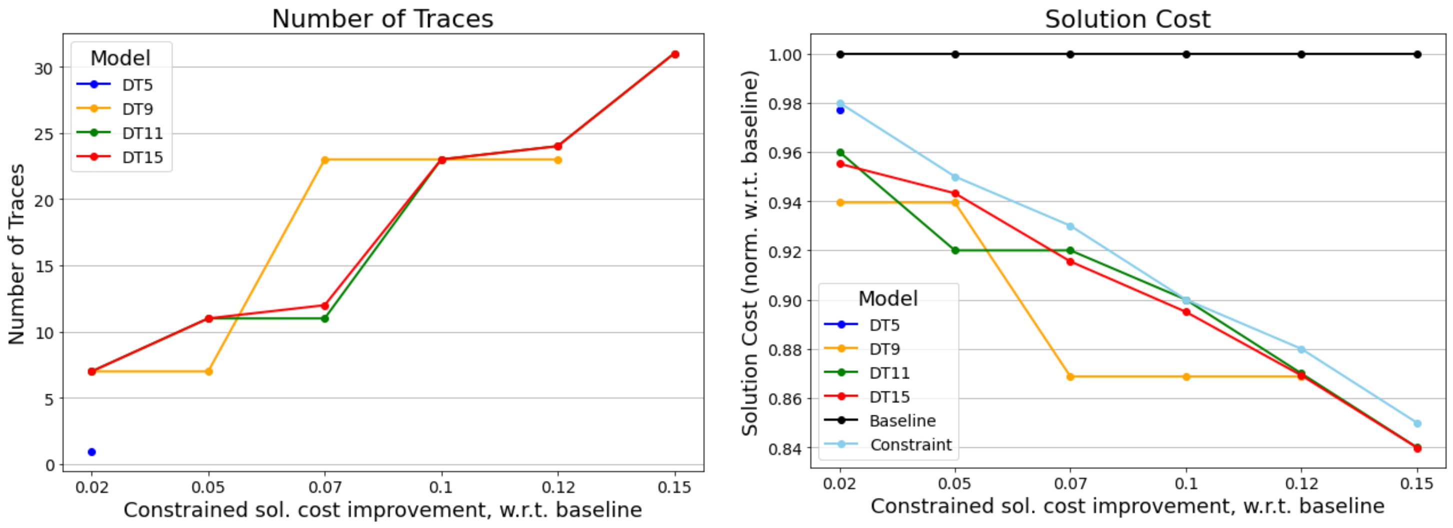
Table 20: Average maximum cost improvement found by each optimizer.

A larger DT depth is beneficial to the quality of the result found by the high-level optimizer: deep trees detect larger improvements in cost compared to shallower ones.

In Graph 16 below we examine only one instance (#13). For the different DTs we visualize how optimization results (i.e. memory, time, cost, and nTraces) change based on the imposed improvement. Memory, time, and cost are normalized by the baseline value, i.e. ratio of the values between the hybrid optimizer and the greedy heuristic. To allow a better visualization we also plot the baseline, whose value is trivial (namely, 1) since variables are reported normalized. In the cost plot we also visualize the constraint value; a cost improvement of *x%* corresponds to a baseline-normalized cost of *(1-x)*, e.g. 2% improvement is equivalent to 0.98. In all plots the cost improvement is reported as decimal value, i.e. *0.x* corresponds to a *x%* improvement.







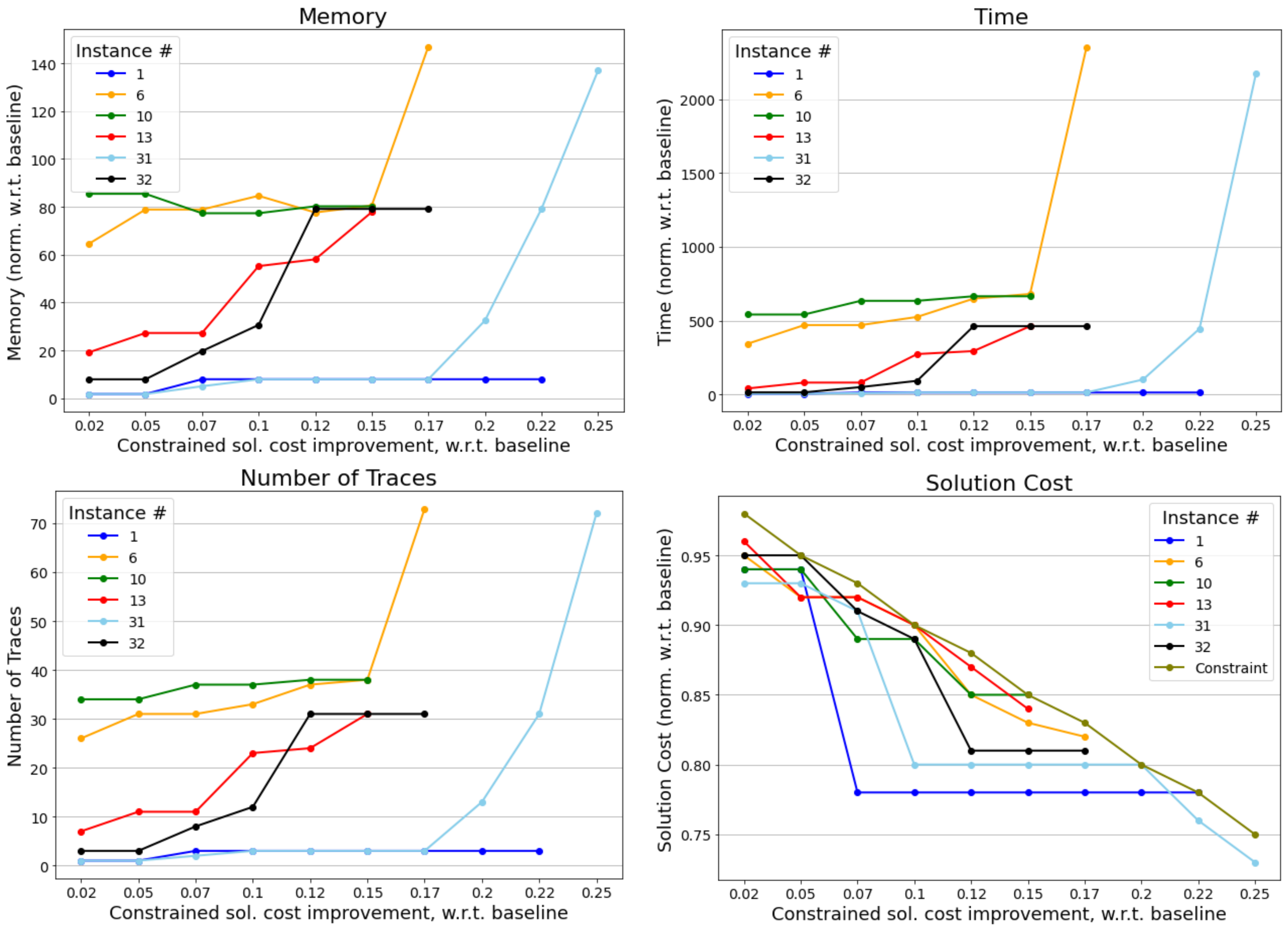
Graph 16: Optimization results (Memory, Time, nTraces, and Cost) for each proposed optimizer on instance #13. Different constraint values are imposed for the cost improvement w.r.t. baseline.

As we would expect, the memory and time suggested by the approach increase as the required cost improvement grows. While the shallowest DT finds an improvement of up to 2%, the deepest tree reaches 15%. The results found by DT_{11} and DT_{15} are very close in value, although DT_{11} stops at 12%, while DT_9 sometimes finds different solutions, e.g. for 5% and 7% improvement. As shown by the cost plot, DT_{15} finds solutions that are closer to the constraint value. This result is not surprising as shallower DTs overfit compared to a deeper DT (i.e. DT_{15}): this results in less precise models that provide coarser predictions compared to DT_{15}.

In Graph 17 we compare optimization results (i.e. memory, time, cost, and nTraces) of DT_{11} on different instances: #1, #6, #10, #13, #31, and #32. For sake of clarity we only examine few instances that are significant; the remaining have similar behaviors.

In general, results in Graph 17 demonstrate that the proposed high-level optimizer is able to tell the user whether a desired cost improvement is feasible and, if it is, to suggest time/memory configurations to reach it and to forecast number of traces and cost.

Using the hybrid offline/online approach allows one to obtain an improvement in the solution value compared to the greedy heuristic, up to 30% for some instances according to the prediction. The instance heavily influences the maximum improvement reachable and the time/memory required. Instance #1 reaches a 22% cost improvement with very little memory/time increase w.r.t. baseline, and the memory/time increase does not change significantly when the improvement grows. Also instance #10 has a constant resource requirement, but it is large (80x increase in memory and 500x in time). Instance #6 and #31 show a constant requirement of memory and time (respectively, large and small) up to a certain improvement, and then they skyrocket. Instances #13 and #32 show a steady increase in the resource as the improvement grows.



Graph 17: Optimization results (Memory, Time, nTraces, and Cost) for the DT$_{11}$-based optimizer on several instances. Different constraint values are imposed for the cost improvement w.r.t. baseline.

Chapter 7
Conclusions

In conclusion, we devise an approach to perform automatic configuration of an algorithm operating on new unseen instances. The proposed method is a combinatorial optimization model that integrates machine learning models via Empirical Model Learning. It is located on top of a hybrid offline/online optimizer, resulting in a two-levels hierarchical system that performs stochastic optimization for the energy management system in a virtual power plant. Results show that our approach allows both automatic decision-making and forecasting on the configuration, online run-time, and computational resources of the low-level algorithm.

The proposed model incorporates information on the behavior of both the underlying controlled VPP and the low-level optimizer. Machine learning techniques are adopted to approximate the behavior of this highly complex system. We use Empirical Model Learning to embed the trained ML models in the combinatorial optimization problem.

By virtue of EML, the optimization model leverages ML to perform decision-making and forecasting over a controller and its controlled system. The high-level optimizer guides the configuration of the low-level one, with no direct communication and with little knowledge of its internal details: the knowledge is given by ML models. Results demonstrate that EML is well-suited for building multi-level optimization systems. By bringing together machine learning and mathematical programming, with EML it is possible to tackle stochastic optimization problems in complex real-world systems.

Integrating machine learning via EML, the optimization model's design is data-driven and automatic; it does not require domain expertise or a hand-crafted modeling process. The proposed system is customizable and interactive, as the user defines constraints and objectives based on the specific use case. Furthermore, it is highly flexible; a variable in the optimization problem can be easily inserted inside constraints or objectives, or it can be predicted based on other variables. This flexibility allows one to perform any desired automatic decision-making and forecasting.

As shown by results, relationships between variables in the system are highly complex and hard to formalize manually. Machine learning techniques are well-suited to capture the knowledge about the system's behavior.

Among the proposed models, artificial neural networks yield poor performance, although they result in smaller optimization problems. The ML model has a larger test set error and training time compared to DTs. The optimization model has a significantly higher solving time, and the quality of its solutions is worse in terms of the ability to find results in different situations.

In decision trees there is a tradeoff between solution quality and times/size related to the model, based on depth. Deeper trees have a larger time for training, embedding, and solving the optimization problem; additionally, the optimization model has a greater memory size and number of variables and constraints. However, the performance of a deeper ML model is better in terms of test error and so is the quality of the solution found by the optimizer; for example, deep DTs allow to have solutions sensible on the specific instance. The suggested DT has a middle depth, 11 in our experiments; it yields comparable solution quality with respect to the deepest tree, with a significantly smaller and less time-consuming optimization model.

The proposed system is ready to be adopted for performing automatic configuration and forecasting on the low-level optimization algorithm in a VPP. In future works it is possible to further experiment with neural networks; their architecture is highly flexible, thus future studies might focus on tuning them to overcome the pitfalls that emerged in our experiments. Another direction for additional studies is to explore the use of random forests as ML models embedded in the combinatorial optimization problem. Finally, since results demonstrate the flexibility of an EML-based system, it is possible to re-use our approach and adopt machine learning to deal with other complex algorithms and real-world optimization problems.

References

[1] E. U. Council and E. U. Parliament, *EU Regulation No 347/2013 on guidelines for trans-European energy infrastructure and repealing,* April 2013.

[2] Q. Yang, T. Yang and W. Li, Smart Power Distribution Systems, Elsevier, 2019.

[3] P. Lombardi, T. Sokolnikova, Z. Styczynski and N. Voropai, "Virtual power plant management considering energy storage systems," in *IFAC*, 2012.

[4] A. De Filippo, M. Lombardi, M. Milano and A. Borghetti, "Robust Optimization for Virtual Power Plants," in *Italian Association for Artificial Intelligence - IAAI*, Nov 2017.

[5] A. De Filippo, M. Lombardi and M. Milano, "Hybrid Offline/Online Optimization Under Uncertainty," in *European Conference on Artificial Intelligence - ECAI*, Santiago de Compostela, Spain, 2020.

[6] W. B. Powell, "A Unified Framework for Optimization Under Uncertainty," in *Optimization Challenges in Complex, Networked and Risky Systems, INFORMS*, 2016.

[7] A. J. Kleywegt, A. Shapiro and T. Homem-de-Mello, "The Sample Average Approximation Method for Stochastic Discrete Optimization," *SIAM Journal on Optimization,* vol. 12, no. 2, p. 479–502, Jan 2002.

[8] B. Verweij, S. Ahmed, A. J. Kleywegt, G. Nemhauser and A. Shapiro, "The Sample Average Approximation Method Applied to Stochastic Routing Problems: A Computational Study," *Computational Optimization and Applications,* vol. 24, p. pages289–333, 2003.

[9] R. Bent and P. V. Hentenryck, Online Stochastic Combinatorial Optimization, The MIT Press, 2006.

[10] P. V. Hentenryck, R. Bent and E. Upfal, "Online stochastic optimization under time constraints," *Annals of Operations Research,* vol. 177, p. 151–183, Jun 2010.

[11] A. B. Philpott and V. L. d. Matos, "Dynamic sampling algorithms for multi-stage stochastic programs with risk aversion," *European Journal of Operational Research*, vol. 218, no. 2, p. 470–483, 2012.

[12] A. De Filippo, M. Lombardi and M. Milano, "Methods for off-line/online optimization under uncertainty," in *International Joint Conferences on Artificial Intelligence - IJCAI*, 2018.

[13] A. De Filippo, M. Lombardi and M. Milano, "How to Tame Your Anticipatory Algorithm," in *International Joint Conference on Artificial Intelligence - IJCAI*, Macao, 2019.

[14] C. Lee and J. Gauvain, "Maximum a posteriori estimation for multivariate Gaussian mixture observations of Markov chains," in *IEEE Transactions on Speech and Audio Processing*, May 1994.

[15] B. Silverman, "Density Estimation for Statistics and Data Analysis," *Monographs on Statistics and Applied Probability*, 2018.

[16] R. Palma-Behnke, C. Benavides, E. Aranda, J. Llanos and D. S´aez, "Energy management system for a renewable based microgrid with a demand side management mechanism," in *IEEE Symposium on Computational Intelligence Applications In Smart Grid - CIASG*, Paris, 2011.

[17] H. Bai, S. Miao, X. Ran and C. Ye, "Optimal dispatch strategy of a virtual power plant containing battery switch stations in a unified electricity market," *Energies*, vol. 8, no. 3, p. 2268–2289, 2015.

[18] A. N. Espinosa and L. N. Ochoa, "Low voltage networks models and low carbon technology profiles," Manchester, June 2015.

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

[20] H. Drucker, C. J. C. Burges, L. Kaufman and A. J. Smola, "Support vector regression machines," in *International Conference on Neural Information Processing Systems - NeurIPS*, Denver, Colorado, 1997.

[21] K. HORNIK, "Approximation Capabilities of Multilayer Feedforward Networks," *Neural Networks*, vol. 4, pp. 251-257, 1991.

[22] M. Leshno, V. Y. Lin, A. Pinkus and S. Schocken, "Multilayer feedforward networks with a nonpolynomial activation function can

approximate any function," *Neural Networks,* vol. 6, no. 6, pp. 861-867, 1993.

[23] D. Yarotsky, "Universal approximations of invariant maps by neural networks," 2018.

[24] S. Ruder, "An overview of gradient descent optimization algorithms," Jun 2017.

[25] D. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," in *International Conference on Learning Representations*, 2014.

[26] M. J. L. O, "Introduction to Radial Basis Function Networks," Edinburgh, Scotland, 1996.

[27] N. Altman, "An Introduction to Kernel and Nearest-Neighbor Nonparametric Regression," *The American Statistician,* vol. 46, no. 3, pp. 175-185, 1992.

[28] R. Kondor, "Regression by linear combination of basis," 2004.

[29] T. Evgeniou and M. Pontil, "Support Vector Machines: Theory and Applications," in *Advanced Course on Artificial Intelligence - ACAI*, Chania, Greece, 2001.

[30] M. A. Hearst, S. T. Dumais, J. P. E. Osuna and B. Scholkopf, "Support vector machines," *IEEE Intelligent Systems and their Applications,* pp. 18-28, 2008.

[31] B. Hammer and K. A. Gersmann, "Note on the Universal Approximation Capability of Support Vector Machines," *Neural Processing Letters,* vol. 17, p. 43–53, 2003.

[32] J. Shlens, "A Tutorial on Principal Component Analysis," Mountain View, California, 2020.

[33] M. Lombardi and M. Milano, "Boosting Combinatorial Problem Modeling with Machine Learning," in *International Joint Conference on Artificial Intelligence - IJCAI*, 2018.

[34] M. Lombardi, M. Milano and A. Bartolini, "Empirical decision model learning," *Artificial Intelligence,* vol. 244, p. 343–367, 2017.

[35] A. Bonfietti, M. Lombardi and M. Milano, "Embedding Decision Trees and Random Forests in Constraint Programming," in *International*

Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research - CPAIOR, 2015.

[36] G. Zhang and W. B. Kleijn, "Training Deep Neural Networks via Optimization Over Graphs," in *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Calgary, AB, 2018.

[37] R. Anderson, J. M. Huchette, C. Tjandraatmadja and J. P. Vielma, "Strong mixed-integer programming formulations for trained neural networks," *Mathematical Programming,* vol. 183, p. 3–39, 2020.

[38] A. Choromanska, M. Henaff, M. Mathieu, G. B. Arous and Y. LeCun, "The Loss Surface of Multilayer Networks," in *International Conference on Artificial Intelligence and Statistics - AISTATS*, 2015.

[39] J. Sola and J. Sevilla, "Importance of input data normalization for the application of neural networks to complex industrial problems," in *IEEE Transactions on IEEE Transactions on*, 1997.

[40] L. Xiao-tong, "Study on Data Normalization in BP Neural Network," *Mechanical Engineering & Automation,* 2010.

[41] D. T.Jayalakshmi, "Statistical Normalization and Back Propagation," *International Journal of Computer Theory and Engineering,* vol. 3, no. 1, 2011.

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

[43] H. Drucker, C. J. C. Burges, L. Kaufman, A. Smola and V. Vapnik, "Support Vector Regression Machines," in *International Conference on Neural Information Processing Systems - NeurIPS,* Denver, Colorado, Dec 1996.

[44] H. Zou, T. Hastie and R. Tibshirani, "Sparse Principal Component Analysis," *Journal of Computational and Graphical Statistics,* vol. 15, no. 2, p. 265–286, 2007.

Appendix A
Dataset Analysis

In section 5.2 we perform an analysis on the dataset used to train machine learning models that are embedded in the high-level optimizer. We report here additional information and plots regarding the dataset.

A.1 All Variables

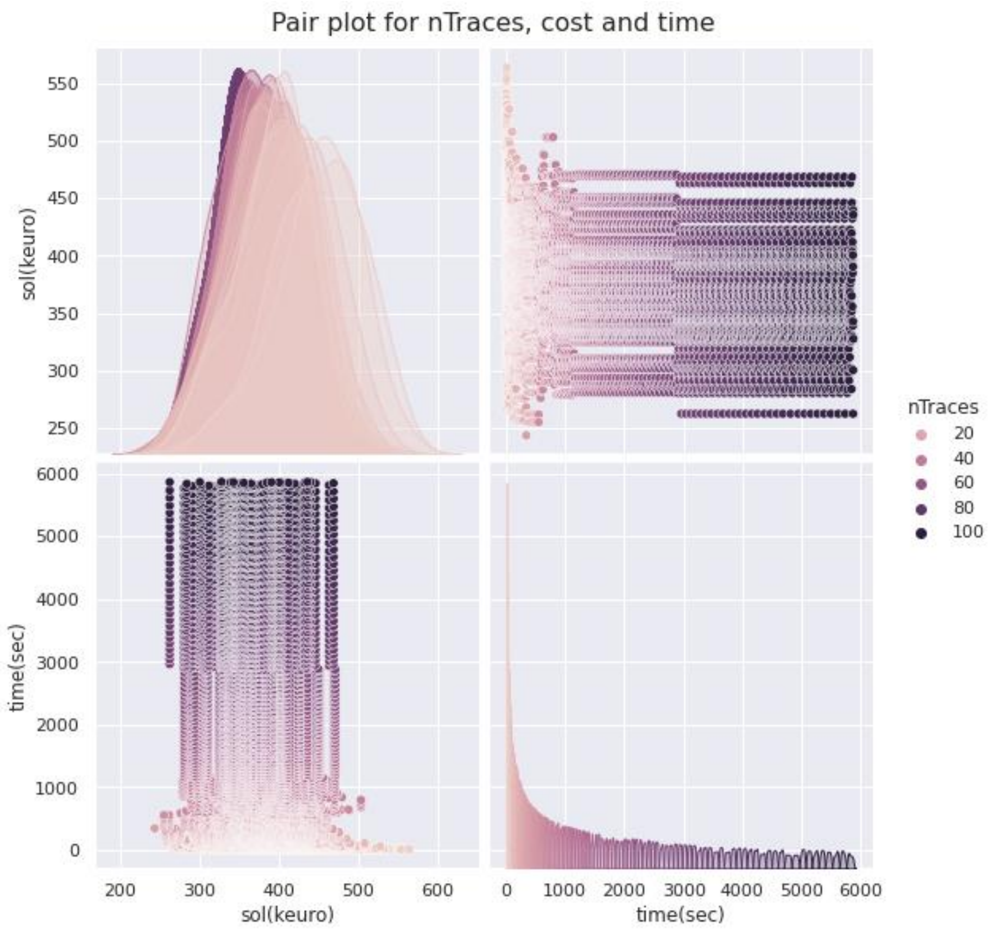
General statistics on the dataset:

	nTraces	Sol Cost (k€)	Sol Cost Norm	Time (sec)	Time Norm	Mem Avg (MB)	Mem Avg Norm	Mem Max (MB)	CPU Avg (%)	CPU Norm	CPU Max (%)
mean	50.5	373.89	0.79	1956.59	1334.67	4545.20	89.28	5576.30	0.26	1.93	1.52
std	28.87	47.28	0.10	1714.41	1183.00	2328.99	46.09	2889.62	0.23	1.70	2.10
min	1.00	243.19	0.51	4.26	3.08	17.08	0.34	90.91	0.00	0.00	0.00
25%	25.75	339.66	0.73	426.62	287.41	2811.98	56.19	3358.76	0.06	0.50	0.30
50%	50.50	372.03	0.79	1470.31	994.90	4596.65	89.19	5618.45	0.24	1.71	0.50
75%	75.25	405.03	0.85	3209.42	2181.75	6348.72	125.11	8014.44	0.37	2.65	2.10
max	100.00	563.83	1.25	5871.14	5045.00	10046.60	201.74	10534.83	1.03	10.10	8.90

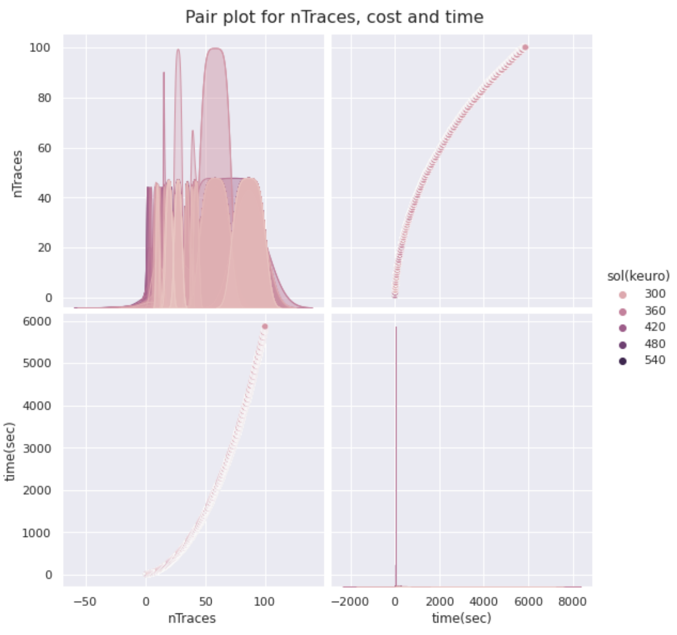
Table 21: Statistics on each column of the dataset containing records of runs of the hybrid offline/online algorithm.

We report in Graph 18 the pairwise plot between time and solution cost, where nTraces is colored. In this situation when one variable is plotted against itself, i.e. diagonal of the grid, we show a Kernel Density Estimate (KDE) layered by the colored variable, i.e. nTraces in this example.

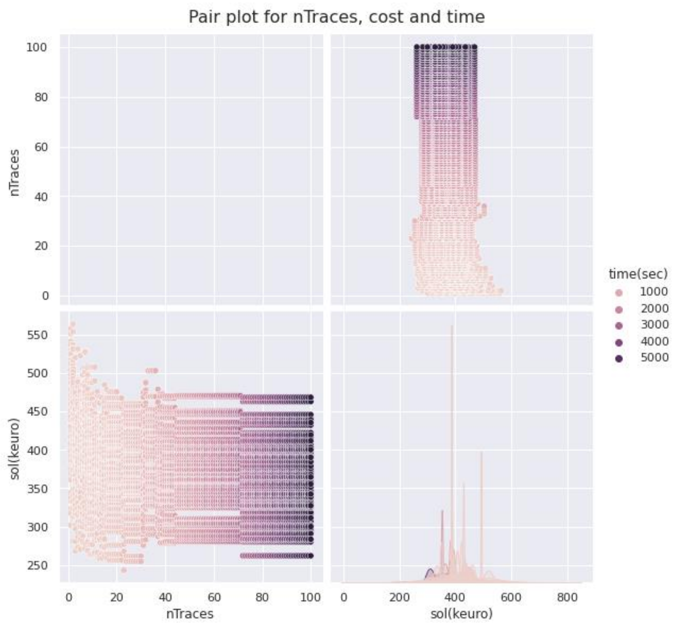
Graph 19 and 20 present similar pairwise plots where variables are, respectively, nTraces and time with cost colored, and nTraces and cost with time colored.



Graph 18: Pair plot for solution cost and time, where the number of traces is colored.



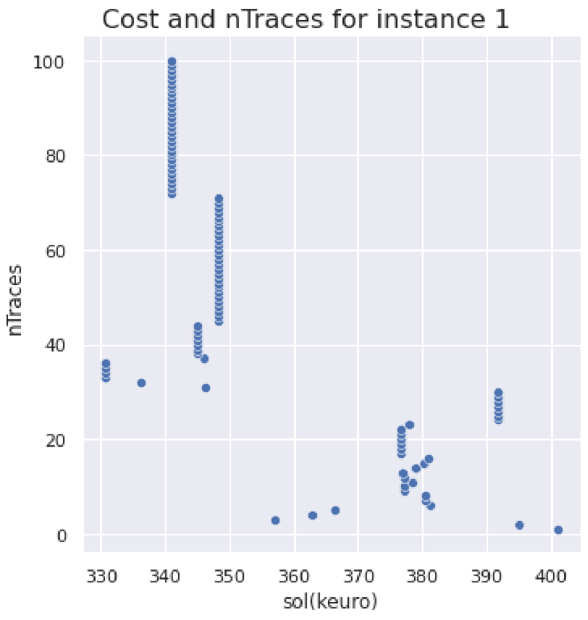
Graph 19: Pair plot for number of traces and resolution time, where the solution cost is colored.



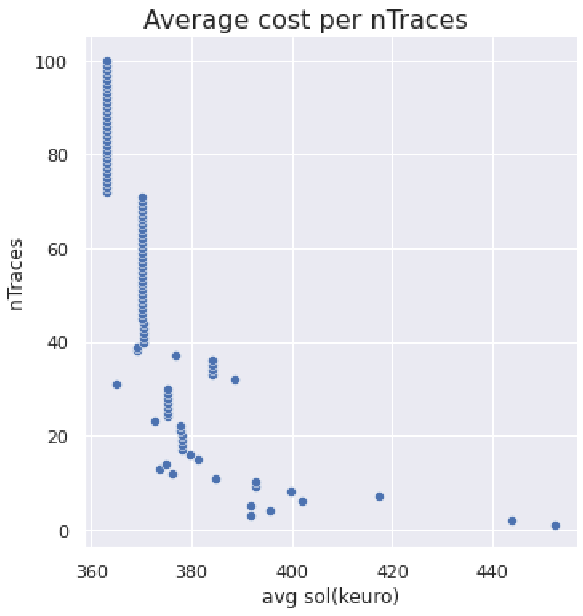
Graph 20: Pair plot for number of traces and solution cost, where the resolution time is colored.

A.2 Number of Traces and Cost

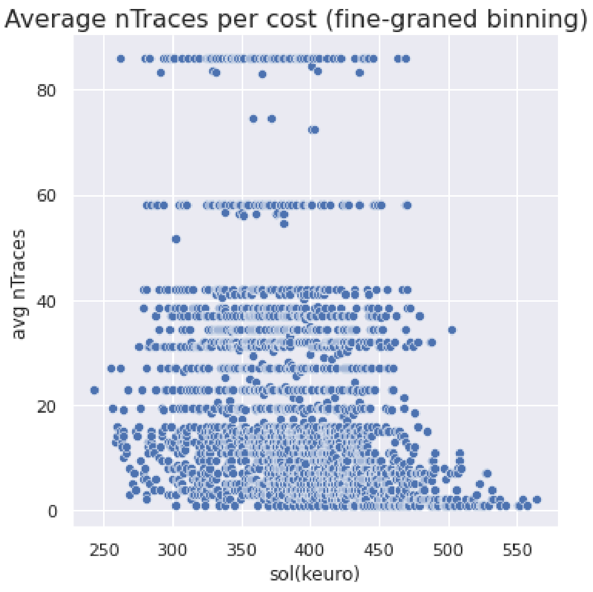
These plots detail the relationship between nTraces and cost.



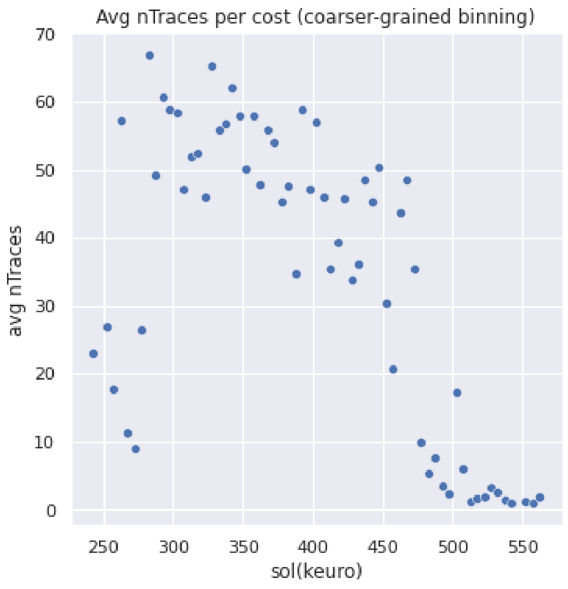
Graph 21: Scatterplot between solution cost and nTraces for one instance (#1).



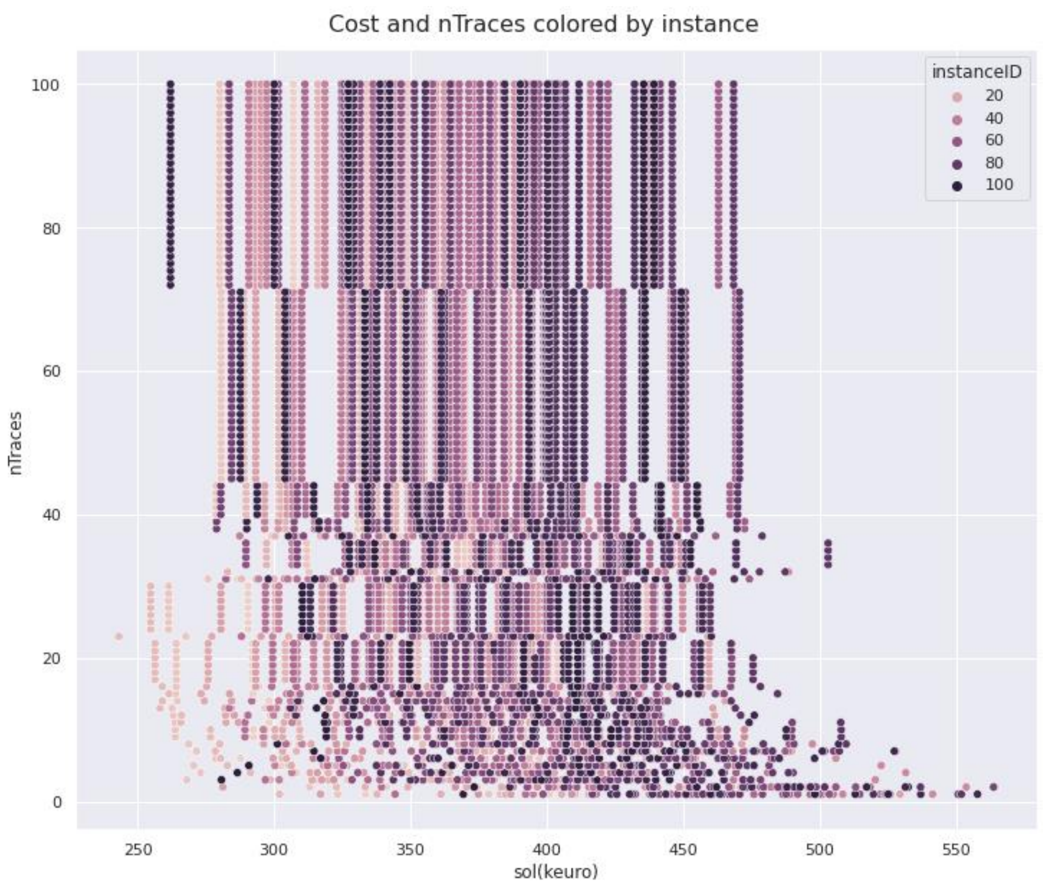
Graph 22: Scatterplot of the average cost per nTraces. For each value of nTraces we average all solution costs.



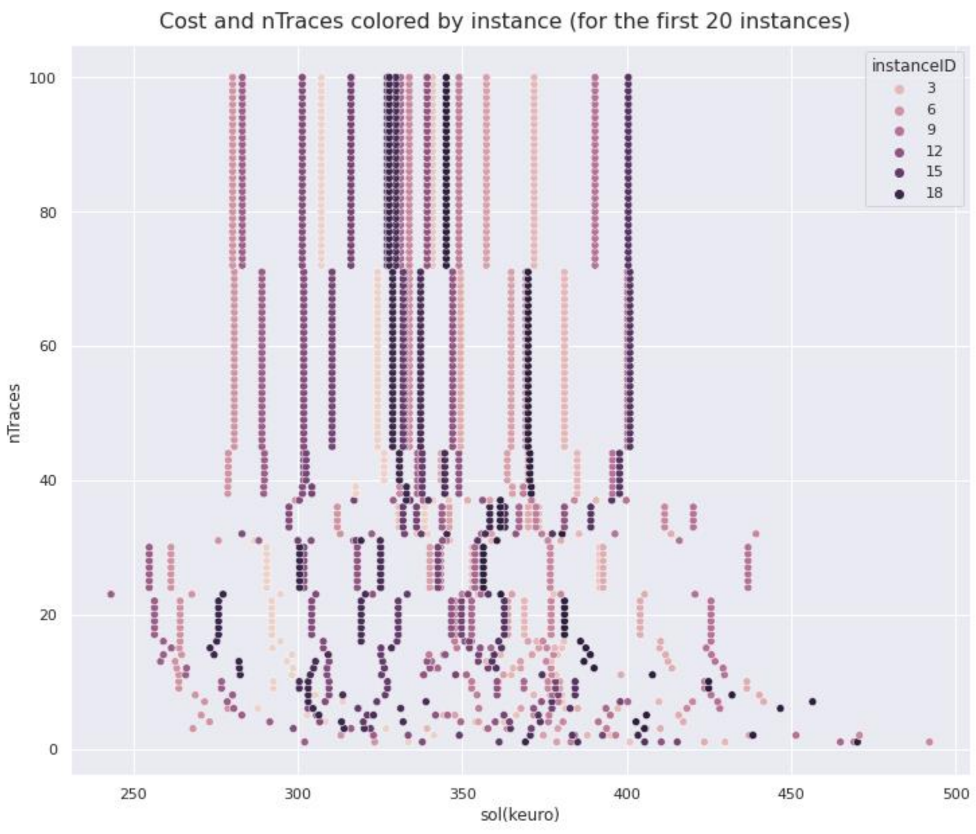
Graph 23: Scatterplot of the average number of traces per cost. For each value of cost we average all number of traces.



Graph 24: Scatterplot of the average number of traces per cost, with binned cost. We perform binning on the cost with a range of 5, i.e. we split the cost's domain in intervals of length 5 and we group together all data points whose cost is within an interval. For each interval we average the number of traces.



Graph 25: Scatterplot of solution cost and number of traces, with the instance id colored.



Graph 26: For the first 20 instances, scatterplot of solution cost and number of traces, with the instance id colored. This helps to shed light on how the instance influences the relationship between cost and nTraces.

Appendix B
Machine Learning

In section 5.3 several machine learning techniques are explored to model the relationship between variables, then the final models are trained and tested. We report here complete results regarding the ML models analyzed while building the high-level optimizer.

B.1 All Models

We leverage different machine learning techniques to capture the relationships between two or more variables. Here reported are results for all ML models experimented, for those sets of variables that were reported partially in section 5.3.1.1.

i. nTraces and cost

Focus on the relationship between number of traces and solution value. Each model predicts nTraces and uses as features either cost, PV, and load or just cost.

Classifiers, using cost, PV, and Load as features:

Model	**Accuracy**
KNN: cross-validation to choose K (K=5 to 8). Best: 5	0.009
SVM classifier: cross-validation to choose the regularization parameter C (C=10-4 to 104, multiplying by 10). Best: 1	0.002

Table 22: Test set performance for classifiers that predict nTraces using cost, PV, and load as features.

Regressors:

Model	**R^2**
Linear regression with change of basis to quadratic base, i.e. polynomial base of degree 2, with cost, PV, and load as features.	0.28744
Linear regression with change of basis to quadratic base, i.e. polynomial base of degree 2, just cost as feature.	0.07482
Random forest regression, 100 trees with max depth 40.	0.88113
Extra Trees regressor, 100 trees with max depth 40.	0.88836
Random Trees Embedding (completely random untrained RF, 5 trees with max depth 6) to generate a feature embedding for PV and Load. Then, use Extra Tree (100 trees, max depth 40) to perform regression taking as features: cost, embedded PV and Load	0.89015
Same as above, using only PV and Load as features.	0.02513
Same as above, using cost and Load as features.	0.88939
Same as above, using PV and cost as features.	0.89083
Same as above, using only cost as features. We note that cost alone achieves a high score, adding PV and Load improves it but not by a large amount.	0.85111
Use an informed (i.e. trained) representation of PV and Load, generated using a trained RF classifier (5 trees with max depth 6). Then use an Extra Tree to perform regression with features cost, embedded PV and Load. There is a slight compared to the uninformed representation.	0.89077
NN regression - architecture (400,), ReLU as activation, Adam optimizer, learning rate 0.001, no validation early stopping, batch size 200, 500 epochs.	0.07606
Use NN classifier - architecture (400,) with 5 final classes – to generate feature embedding for PV and Load. Then, similarly to above, use an Extra Tree to perform regression with features cost, embedded PV and Load.	0.85102
Use PCA (with 5 p.c.) to generate feature embedding for PV and Load. Then apply Extra Trees Regressor as above with reduced PV, Load and cost.	0.88972
Support Vector Machine regressor with RBF kernel, regularization parameter C=100.	0.90514

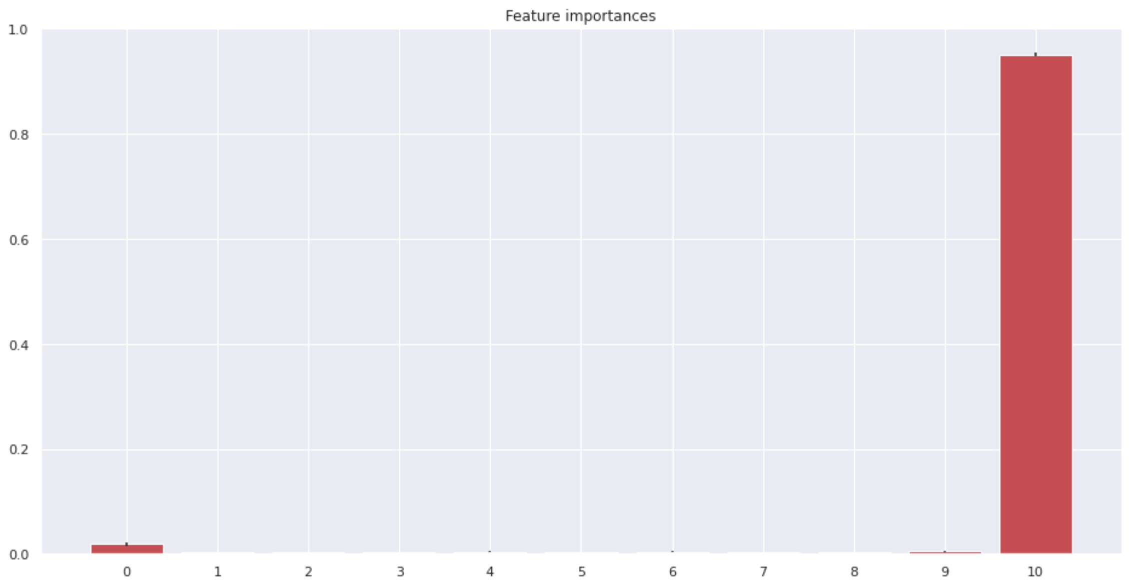
Table 23: Test set performance for regressors that predict nTraces using cost and (if applicable) PV/load as features.

ii. nTraces and memory

Models that predicts the number of traces using as features either average memory, PV, and load or just the average memory.

Model	**R^2**
Linear regressor Lasso, all features.	0.65789
Linear regression Lasso with polynomial basis degree 2, all features.	0.65216
Linear regressor Lasso, just memory as feature.	0.64951
Linear regression Lasso with polynomial basis degree 3, just memory as feature.	0.66070
Random forest regressor, maximum depth 20, all features.	0.93232
RF regressor (maximum depth 20), preceded by PCA (5 p.c.) for dim reduction of PV and Load.	0.94614
RF regressor (maximum depth 20), preceded by RF Classifier for dim reduction of PV and Load. As shown by features' importance in Graph 27 the prediction is completely based on memory and it ignores PV and Load.	0.94722
Extra Tree Regressor, 100 trees with maximum depth 40, all features.	0.83576
SVR with RBF kernel, all features.	0.00005
SVR with linear kernel, all features.	0.76906

Table 24: Test set performance for regressors that predict nTraces using average memory and (if applicable) PV/load as features.



Graph 27: Feature importance for the RF regressor that takes as features PV (0-4), Load (5-9), and average memory (10). A RF Classifier is used beforehand for dimensionality reduction of both PV and Load.

iii. Total

Unified models:

Model	R^2
Extra Trees regressor (maximum depth 20), using average memory.	0.99999
Extra Trees regressor (maximum depth 20), using maximum memory.	0.99999
PCA (with 5 p.c.) for dim. reduction of PV and Load. Then, Extra Trees regressor (maximum depth 20), using average memory.	0.99999
NN with architecture [280], activation tanh, maximum epochs 500, using average memory.	0.98165
PCA (with 5 p.c.) for dim. reduction of PV and Load. Then, NN with architecture [280], activation tanh, maximum epochs 500, using average memory.	0.98555
SVR with RBF kernel, using average memory.	0
SVR with linear kernel, using average memory.	0.94796

Table 25: Test set performance for regressors that predict nTraces using all remaining variables as features. The model is unified, i.e. a unique regressor takes all features and predicts nTraces.

B.2 Decision Trees

In this section we report complete results for experiments focused on the decision trees in Section 5.3.1.2. We experiment with 4 models on 3 forms of data normalization (original, standardized, and min-max scaled).

Results for models that do not use PV/Load as features:

	Model						
	nTraces		**Cost**		**Time**	**Average memory**	
Original							
DT depth bound	1	No	No	5	No	1	No
DT depth	1	7	54	5	24	1	26
R^2	0.7525	1.0000	-0.5038	0.0979	1.0000	0.5648	0.9973
Standardized							
DT depth bound	1	No	No	5	No	1	No
DT depth	1	7	54	5	22	1	23
R^2	0.7525	1.0000	-0.4925	0.0979	1.0000	0.5648	0.9980

Min-max feature scaling							
DT depth bound	1	No	No	5	No	1	No
DT depth	1	7	54	5	21	1	23
R^2	0.7525	1.0000	-0.4748	0.0979	1.0000	0.5648	0.9976

Table 26: Test set performance on all dataset's normalizations for experimental DTs that do not use PV/Load as features.

Results for models that use PV/Load as features:

	Model						
	nTraces		**Cost**		**Time**	**Average memory**	
Original							
DT depth bound	1	No	No	5	No	1	No
DT depth	1	7	21	5	22	1	27
R^2	0.7525	1.0000	0.9391	0.5219	1.0000	0.5648	0.9882
Standardized							
DT depth bound	1	No	No	5	No	1	No
DT depth	1	7	21	5	17	1	22
R^2	0.7525	1.0000	0.9431	0.5219	1.0000	0.5648	0.9914
Min-max feature scaling							
DT depth bound	1	No	No	5	No	1	No
DT depth	1	7	21	5	13	1	22
R^2	0.7525	1.0000	0.9446	0.5219	1.0000	0.5648	0.9920

Table 27: Test set performance on all dataset's normalizations for experimental DTs that use PV/Load as features.

Appendix C

Final ML Models

Here reported are complete information about the architecture and hyperparameters selected for neural networks in section 5.3.2.2. These models are the ones embedded into the combinatorial optimization model used in optimization experiments.

Cross-validation is used to select hyperparameters in the neural network among:

- *Arch*: The architecture among:
  - [100, 100], "2" in the table below.
  - [100, 100, 100], "3" in the table below.
  - [100, 100, 100, 100], "4" in the table below.
- *Act*: The activation between ReLU (R) and tanh (T).
- *BS*: Batch size among 100, 200, and 400.
- *Ep*: Epochs among 50, 100, and 500.

Hyperparameters selected for neural networks are the following:

Features	**Data Norm.**	**Model**											
		Cost				**Time**				**Average Memory**			
		Arch	**Act**	**BS**	**Ep**	**Arch**	**Act**	**BS**	**Ep**	**Arch**	**Act**	**BS**	**Ep**
nTraces	**Orig.**	3	R	200	50	3	R	100	100	4	R	200	100
	Std.	3	R	100	100	**2**	**R**	**200**	**100**	**4**	**R**	**100**	**100**
	M.m.s.	4	R	100	100	2	R	200	50	4	R	100	100
nTraces, PV and Load	**Orig.**	3	R	100	100	3	R	100	100	4	R	100	100
	Std.	**4**	**R**	**100**	**100**	2	R	100	100	4	T	100	100
	M.m.s.	4	R	100	100	4	R	100	50	4	R	100	100

Table 28: Hyperparameters selected for the NN models. Some of them (in blue) are embedded into the combinatorial optimization problem and used in optimization experiments.

Appendix D
Comparative Experiments

We report complete results for the comparative experiments detailed in section 6.1. An additional problem is under analysis: as a sanity check, we minimize time, memory or solution value for a fixed number of traces given as constraint; results should be similar with all the three objectives. This is the last set of experiments reported in the table. Times are in seconds.

Objective (minimized)	Constraints	Variable	Empirical ML Model							
			DT 9		DT 11		DT 15		NN	
			Value	Res. Time	Value	Res. time	Value	Res. time	Value	Res. time
nTraces	Cost <= 398	nTraces	1	0.2191	1	1.5096	1	5.1944	No solution	1200
		Mem	88.92		88.92		88.92			
		Time	6.93		6.93		9.51			
	Cost <= 279	nTraces	3	0.1924	3	1.2222	3	6.0768	No solution	1200
		Mem	396.82		396.82		396.82			
		Time	18.39		18.39		18.39			
	Cost <= 256	nTraces	No solution	0.1494	24	1.2049	24	6.2743	No solution	1200
		Mem			2953.95		2953.95			
		Time			381.99		381.99			
	Cost <= 516	nTraces	1	0.2554	1	1.4513	1	5.2468	2	151.9691
		Mem	88.92		88.92		88.92		70.97	
		Time	6.93		6.93		9.51		5.56	
Cost	Mem <= 3963, Time <= 2128	Cost	255.91	0.2724	254.83	2.0288	254.83	6.6248	453.06	1200
		nTraces	13		24		24		37	
	Mem <= 819, Time <= 71	Cost	268.10	0.2346	268.10	1.1606	268.10	4.3203	337.06	1200
		nTraces	3		3		3		8	
	Mem <= 586, Time <= 71	Cost	268.10	0.2331	268.10	1.0574	268.10	4.3037	283.68	1200
		nTraces	3		3		3		6	

	Mem <= 9203, Time <= 71	Cost	281.29	0.2253	281.29	0.9138	281.29	3.8255	394.42	1200
		nTraces	1		1		1		2	
	Mem <= 9203, Time <= 5385	Cost	255.91	0.2760	254.83	1.6010	254.83	6.1637	317.07	1200
		nTraces	13		24		24		50	
Time	-	Time	6.93	0.2484	6.93	1.3874	9.51	5.6524	5.56	363.0372
		nTraces	1		1		1		2	
	Cost <= 397	Time	6.93	0.2190	6.93	1.6775	9.51	5.3994	5.56	593.4029
		nTraces	1		1		1		2	
	Cost <= 397, Time >= 1099	Time	1136.16	0.2130	1136.16	0.9668	1136.16	4.3337	1138.68	78.8298
		nTraces	44		44		44		44	
	Time >= 1099	Time	1001.86	0.2494	1001.86	1.0806	1001.86	4.2975	1001.07	34.3314
		nTraces	41		41		41		41	
	Cost <= 256	Time	No solution	0.1835	381.99	1.2097	381.99	5.8430	No solution	1200
		nTraces			24		24			
	Cost <= 516	Time	6.93	0.2421	6.93	1.5417	9.51	5.8043	5.56	264.3807
		nTraces	1		1		1		2	
Mem	Cost <= 398	Mem	88.92	0.2173	88.92	1.6402	88.92	5.4473	70.97	1200
	Cost <= 256	Mem	No solution	0.1493	2953.95	1.1909	2953.95	5.8295	6166.68	1200
		nTraces			24.00000		24.00000		77	
	Cost <= 516	Mem	88.92	0.2426	88.92	1.5467	88.92	5.5582	70.97	1200
Mem	nTraces == 25	Mem	3096.36	0.0705	3096.36	0.5273	3096.36	3.4764	2976.92	13.9138
		Time	411.11		411.11		411.11		409.94	
		Cost	261.44		261.44		386.55		443.02	
Time		Mem	3096.36	0.0838	3096.36	0.5190	3096.36	3.3477	2444.10	13.8680
		Time	411.11		411.11		411.11		802.16	
		Cost	261.44		261.44		386.55		443.02	
Cost		Mem	3096.36	0.0802	3096.36	0.5374	3096.36	3.4372	1892.42	1179.835
		Time	411.11		411.11		411.11		409.94	
		Cost	255.91		254.83		254.83		243.19	

Table 29: Complete comparative experiments for DT_9, DT_{11}, DT_{15}, NN. For each problem we report objective and constraints, resolution time for the high-level optimizer and, for each variable of interest, the solution value.