A learning-based mathematical programming formulation for the automatic configuration of optimization solvers *

Gabriele Iommazzo[1,2], Claudia D'Ambrosio[1], Antonio Frangioni[2], and Leo Liberti[1]

[1] LIX CNRS, École Polytechnique, Institut Polytechnique de Paris, Palaiseau, France,
{dambrosio,giommazz,liberti}@lix.polytechnique.fr
[2] Dip. di Informatica, Università di Pisa, Pisa, Italy,
frangio@di.unipi.it

Abstract. We propose a methodology, based on machine learning and optimization, for selecting a solver configuration for a given instance. First, we employ a set of solved instances and configurations in order to learn a performance function of the solver. Secondly, we formulate a mixed-integer nonlinear program where the objective/constraints explicitly encode the learnt information, and which we solve, upon the arrival of an unknown instance, to find the best solver configuration for that instance, based on the performance function. The main novelty of our approach lies in the fact that the configuration set search problem is formulated as a mathematical program, which allows us to a) enforce hard dependence and compatibility constraints on the configurations, and b) solve it efficiently with off-the-shelf optimization tools.

Keywords: automatic algorithm configuration, mathematical programming, machine learning, optimization solver configuration, hydro unit committment

1 Introduction

We address the problem of finding instance-wise optimal configurations for general Mathematical Programming (MP) solvers. We are particularly motivated by state-of-the-art general-purpose solvers, which combine a large set of diverse algorithmic components (relaxations, heuristics, cutting planes, branching, ...) and therefore have a long list of user-configurable parameters; tweaking them can have a significant impact on the quality of the obtained solution and/or on the efficiency of the solution process (see, e.g., [**?**]). Good solvers have effective default parameter configurations, carefully selected to provide good performances

* This paper has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Sklodowska-Curie grant agreement n. 764759 "MINOA".

in most cases. Furthermore, tuning tools may be available (e.g., [**?**, Ch. 10]) which run the solver, with different configurations, on one or more instances within a given time limit, and record the best parameter values encountered. Despite all this, the produced parameter configurations may still be highly suboptimal with specific instances. Hence, a manual search for the best parameter values may be required. This is a highly nontrivial and time-consuming task, due to the large amount of available parameters (see, e.g., [**?**]), which requires a profound knowledge of the application at hand and an extensive experience in solver usage. Therefore, it is of significant interest to develop general approaches, capable of performing it efficiently and effectively in an automatic way.

This setting is an instance of the Algorithm Configuration Problem (ACP) [**?**], which is defined as follows: given a target algorithm, its set of parameters, a set of instances of a problem class and a measure of the performance of the target algorithm on a pair (instance, algorithmic configuration), find the parameter configuration providing optimal algorithmic performance according to the given measure, on a specific instance or instance set. Several domains can benefit from automating this task. Some possible applications, beyond MP solvers (see, e.g., [**?**] and references therein), are: solver configuration for the propositional satisfiability problem, hyperparameter tuning of ML models or pipelines, algorithm selection, administering ad-hoc medical treatment, etc.

Our approach for addressing the ACP on MP solvers is based on a two-fold process:

(i) in the *Performance Map Learning Phase* (PMLP), supervised Machine Learning (ML) techniques [**?**] are used to learn a *performance function*, which maps some features of the instance being solved, and the parameter configuration, into some measure of solver efficiency and effectiveness;
(ii) the formal model underlying the ML methodology used in the PMLP is translated into MP terms; the resulting formulation, together with constraints encoding the compatibility of the configuration parameter values, yields the *Configuration Set Search Problem* (CSSP), a Mixed-Integer Nonlinear Program (MINLP) which, for a given instance, finds the configuration providing optimal performance with respect to the performance function.

The main novelty of our approach lies in the fact that we explicitly model and optimize the CSSP using the mathematical description of the PMLP technique. This is in contrast to most of the existing ACP approaches, which instead employ heuristics such as local searches [**?**,**?**], genetic algorithms [**?**], evolutionary strategies [**?**] and other methods [**?**]. Basically, most approaches consider the performance function as a black box, even when it is estimated by means of some ML technique and, therefore, they cannot reasonably hope to find a global minimum when the number of parameters grows. Rather, one of the strengths of our methodology is that it exploits the mathematical structure of the CSSP, solving it with sophisticated, off-the-shelf MP solvers. Moreover, formulating the CSSP by MP is advantageous as it allows the seamless integration of the compatibility constraints on the configuration parameters, which is something that other ACP

methods may struggle with. The idea of using a ML predictor to define the unknown components (constraints, objective) of a MP has been already explored in *data-driven optimization*. In general, it is possible to represent the ML model of a mapping/relation as a MP (or, equivalently, a Constraint Programming model) and optimize upon this [**?**]; however, while this is in principle possible, the set of successful applications in practice is limited. Indeed, using this approach in the ACP context is, to the best of our knowledge, new; it also comes with some specific twists. We tested this idea with the following components: we configured nine parameters of the IBM ILOG CPLEX solver [**?**], which we employed to solve instances of the Hydro Unit Commitment (HUC) problem [**?**], we chose Support Vector Regression (SVR) [**?**] as the PMLP learning methodology, and we used the off-the-shelf MINLP solver Bonmin [**?**] to solve the CSSP.

The paper is structured as follows: in Sec. **??** we will review existing work on algorithm configuration; in Sec. **??** we will detail our approach and provide the explicit formulation of the CSSP with SVR; in Sec. **??** we will discuss some computational results.

2 The algorithm configuration problem

Most algorithms have a very high number of configurable parameters of various types (boolean, categorical, integer, continuous), which usually makes the ACP very hard to solve in practice. Notably, this issue significantly affects MP solvers: they are highly complex pieces of software, embedding several computational components that tackle the different phases of the solution process; the many available algorithmic choices are exposed to the user as a long list of tunable parameters (for example, more than 150 in CPLEX [**?**]).

Approaches to the ACP can be compared based on how they fit into the following two categories: Per-Set (PS) or Per-Instance (PI); offline or online. In PS approaches, the optimal configuration is defined as the one with the best overall performance over a set of instances belonging to the same problem class. Therefore, PS approaches first find the optimal configuration for a problem class and then use it for any instance pertaining to that class. The exploration of the configuration set is generally conducted by means of heuristics, such as various local search procedures [**?**,**?**], racing methods [**?**], genetic algorithms [**?**] or other evolutionary algorithms [**?**]. In this context, an exception is, e.g., the approached described in [**?**], which predicts the performance of the target algorithm by random forest regression and then uses it to guide the sampling in an iterative local search. PS approaches, however, struggle when the target algorithm performance varies considerably among instances belonging to the same problem class. In these cases, PI methodologies, which assume that the optimal algorithmic configuration depends on the instance at hand, are likely to produce better configurations. However, while PS approaches are generally problem-agnostic, PI ones require prior knowledge of the problem at hand, to efficiently encode each instance by a set of features. PI approaches typically focus on learning a good surrogate map of the performance function, generally by performing regression:

this approximation is used to direct the search in the configuration set. In [**?**], for example, linear basis function regression is used to approximate the target algorithm runtime, which is defined as a map of both features and configurations; then, for a new instance with known features, the learnt map is evaluated at all configuration points in an exhaustive search, to find the estimated best one. However, other approaches may be used: in [**?**], for example, a map from instance features to optimal configuration is learnt by a neural network; in [**?**] the ACP is restricted to a single binary parameter of CPLEX, and a classifier is then trained to predict it. In [**?**], instead, CPLEX is run on a given instance for a certain amount of computational resources, then a ranking ML model is trained, on-the-fly, to learn the ordering of branch and bound variables, and it is then used to predict the best branching variable, at each node, for the rest of the execution. Another approach, presented in [**?**], first performs clustering on a set of instances, then uses the PS methodology described in [**?**] to find one good algorithmic configuration for each cluster. In [**?**], instead, instances are automatically clustered in the leaves of a trained decision tree, which also learns the best configuration for each leaf; at test time, a new instance is assigned to a leaf based on its features, and it receives the corresponding configuration. The purpose of an ACP approach is to provide a good algorithmic configuration upon the arrival of an unseen instance. We call a methodology offline if the learning happens before that moment, which is the case for all the approaches cited above. Otherwise, we call an ACP methodology online; these approaches normally use reinforcement learning techniques (see, e.g., [**?**,**?**]) or other heuristics [**?**].

In our approach, we define the performance of the target algorithm as a function of both features and controls, in order to account for the fact that the best configuration of a solver may vary among instances belonging to the same class of problems; this makes our approach PI. Moreover, we perform the PMLP only once, offline, which allows us to solve the resulting CSSP for any new instances in a matter of seconds. What makes our approach stand out from other methodologies is that the learning phase is treated as white-box: the prediction problem of the PMLP is formulated as a MP, which conveniently allows the explicit embedding of a mathematical encoding of the estimated performance into the CSSP, as its objective/constraints. This is opposed to treating the learned predictor as a black-box, and therefore using it as an oracle in brute-force searches or similar heuristics, that typically do not scale as well as optimization techniques.

3 The PMLP and the CSSP

Let $\mathcal{A}$ be the target algorithm, and:

- $\mathcal{C}_{\mathcal{A}}$ be the set of feasible configurations of $\mathcal{A}$. We assume that each configuration $c \in \mathcal{C}_{\mathcal{A}}$ can be encoded into a vector of binary and/or discrete values representing categorical and numerical parameters, and $\mathcal{C}_{\mathcal{A}}$ can be described by means of linear constraints;
- $\mathit{\Pi}$ be the problem to be solved, consisting of an infinite set of instances, and $\mathit{\Pi}' \subset \mathit{\Pi}$ be the (finite) set of instances used for the PMLP;

- F_{Π} be the set of feature vectors used to describe instances, encoded by vectors of continuous or discrete/categorical values (in the latter case they are labelled by reals);
- $p_{\mathcal{A}} : F_{\Pi} \times \mathcal{C}_{\mathcal{A}} \longrightarrow \mathbb{R}$ be the performance function which maps a pair (f, c) (instance feature vector, configuration) to the outcome of an execution of $\mathcal{A}$ (say in terms of the integrality gap reported by the solver after a time limit, but other measures are possible).

With the above definitions, the PMLP and the CSSP are detailed as follows.

3.1 Performance Map Learning Phase

In the PMLP we use a supervised ML predictor, e.g., SVR, to learn the coefficient vector θ^* providing the parameters of a prediction model $\bar{p}_{\mathcal{A}}(\cdot, \cdot, \theta) : F_{\Pi} \times \mathcal{C}_{\mathcal{A}} \rightarrow \mathbb{R}$ of the performance function $p_{\mathcal{A}}(\cdot, \cdot)$. The training set for the PMLP is

$$\mathcal{S} = \big\{(f_i, c_i, p_{\mathcal{A}}(f_i, c_i)) \mid i \in \{1 \dots s\}\big\} \subseteq F_{\Pi'} \times \mathcal{C}_{\mathcal{A}} \times \mathbb{R}, \tag{1}$$

where $s = |\mathcal{S}|$ and the training set labels $p_{\mathcal{A}}(f_i, c_i)$ are computed on the training vectors (f_i, c_i). The vector θ^* is chosen as to hopefully provide a good estimate of $p_{\mathcal{A}}$ on points that do not belong to $\mathcal{S}$, with the details depending on the selected ML technology.

3.2 Configuration Space Search Problem

For a given instance f and parameter vector θ, $\mathsf{CSSP}(f, \theta)$ is the problem of finding the configuration with best estimated performance $\bar{p}_{\mathcal{A}}(f, c, \theta)$:

$$\mathsf{CSSP}(f, \theta) \equiv \min_{c \in \mathcal{C}_{\mathcal{A}}} \bar{p}_{\mathcal{A}}(f, c, \theta)\,. \tag{2}$$

The actual implementation of $\mathsf{CSSP}(f, \theta)$ depends on the MP formulation selected to encode $\bar{p}_{\mathcal{A}}$, which may require auxiliary variables and constraints to define the properties of the ML predictor. If $\bar{p}_{\mathcal{A}}$ yields an accurate estimate of $p_{\mathcal{A}}$, we expect the optimum c^*_{cssp} of $\mathsf{CSSP}(f, \theta)$ to be a good approximation of the true optimal configuration c^* for solving f. However, we remark that a) $\mathsf{CSSP}(f, \theta)$ can be hard to solve, and b) it needs to be solved quickly (otherwise one might as well solve the instance f directly). Hence, incurring the additional computational overhead for solving the CSSP may be advantageous only when the instance at hand is "hard". Achieving a balance between PMLP accuracy and CSSP cost is one of the challenges of this research.

4 Experimental results

We tested our approach on 250 instances of the HUC problem and on 9 parameters of CPLEX, version 12.7. The PMLP and CSSP experiments were conducted on an Intel Xeon CPU E5-2620 v4 @ 2.10GHz architecture, while CPLEX was run on an Intel Xeon Gold 5118 CPU @ 2.30GHz. The pipeline was implemented in Python 3.6.8 [?] and AMPL Version 20200110 [?]. In the following, we detail the algorithmic set-up that we employed.

4.1 Building the dataset

1. *Features.* The HUC is the problem of finding the optimal scheduling of a pump-storage hydro power station, where the commitment and the power generation of the plant must be decided in a short term period, in which inflows and electricity prices are previously forecast. The goal is to maximize the revenue given by power selling (see, e.g., [**?**]). The time horizon is fixed to 24h and the underlying hydro system is also fixed, so that all the instances have the same size. Thus, only 54 elements that vary from day to day are features: the date, 24 hourly prices, 24 hourly inflows, initial and target water volumes, upper and lower bound admitted on the water volumes. We encode them in a vector f of 54 continuous/discrete components. All the instances have been randomly generated with an existing generator that accurately reproduces realistic settings.
2. *Configuration parameters.* Thanks to preliminary tests, we select a subset of 9 discrete CPLEX parameters (`fpheur`, `dive`, `probe`, `heuristicfreq`, `startalgorithm` and `subalgorithm` from `mip.strategy`; `crossover` from `barrier`; `mircuts` and `flowcovers`, from `mip.cuts`), for each of which we consider between 2 and 4 different values. We then combine them so as to obtain 2304 parameter configurations. A configuration is encoded by a vector $c \in \{0, 1\}^{23}$, where each categorical parameter is represented by its incidence vector.
3. *Performance measure.* We use the integrality gap to define $p_{\mathcal{A}}(f, c)$. It has been shown that MIP solvers can be affected by performance variability issues (see, e.g., [**?**]), due to executing the solver on different computing platforms, permuting rows/columns of a model, adding valid but redundant constraints, performing apparently neutral changes to the solution process, etc. In order to tackle this issue, first we sample three different random seeds. For each instance feature vector f and each configuration c, we then carry out the following procedure: (i) we run CPLEX (using the Python API) 3 times on the instance, using the different random seeds, for 60 seconds; (ii) we record the middle out of the three obtained performance values, to be assigned to the pair (f, c). At this point, our dataset contains $250 \times 2304 = 576000$ records. The performance measure thus obtained from CPLEX output, which we call $p_{\texttt{cpx}}(f, c)$, usually contains some extremely large floating point values (e.g., whenever the CPLEX gap has a value close to zero in the denominator), which unduly bias the learning process. We deal with this issue as follows: we compute the maximum $\bar{p}_{\texttt{cpx}}$, over all values of (the range of) $p_{\texttt{cpx}}$, lower than a given threshold (set to 1e+5 in our experiments), re-set all values of $p_{\texttt{cpx}}$ larger than the threshold to $\bar{p}_{\texttt{cpx}} + 100$, then rescale $p_{\texttt{cpx}}$ so that it lies within the interval $[0, 1]$. The resulting performance measure, which in the following we call $p_{\texttt{ml}}(f, c)$, is also the chosen PMLP label. We save both $p_{\texttt{cpx}}$ and $p_{\texttt{ml}}$ in our dataset. We remark that setting the time-limit, imposed on CPLEX runs, to 60 seconds provides the solver enough time to move past the preliminary processing and to begin working on closing the gap, even for very hard instances (i.e., the ones with

long pre-processing times); this allows us to measure the actual impact that different parameter configurations have on the chosen performance measure.

4. *Feature engineering.* We process the date in order to extract the season, the week-day, the year-day, two flags called `isHoliday` and `isWeekend`, and we perform several sine/cosine encodings, that are customarily used to treat cyclical features. Moreover, we craft new features by computing statistics on the remaining 54 features. This task takes around 12 minutes to complete for the whole data set.
5. *Splitting the dataset.* We randomly divide the instances into 187 In-Sample (IS) and 63 Out-of-Sample (OS), and split the dataset rows accordingly (430848 IS and 145152 OS). We use the IS data to perform Feature Selection (FS) and to train the SVR predictor; then, we assess the performance of the PMLP-CSSP pipeline both on OS instances, to test its generalization capabilities to unseen input, and on IS instances, to evaluate its performance on the data that we learn from, as detailed below.
6. *Feature selection.* We use Python's `Pandas DataFrame`'s `corr` function to perform Pearson's Linear Correlation (LC) and `sklearn RandomForestRegressor`'s `feature_importances_` attribute to perform decision trees' Feature Importance (FI), in order to get insights on which features contribute the most to yield accurate predictions. A detailed explanation of the employed FS techniques falls outside of the scope of this document, but the interested reader can refer to [**?**] (for LC) and [**?**, Ch. 10.13,15.3] (for FI), among many others. In the following, we use the shorthand "variables" to refer to the whole list of columns of the learning dataset. In order to perform FS, we use a dedicated subset of the IS dataset, composed of 19388 records and only employed for this task; performing the selected FS techniques on this dataset takes around 8 minutes, and reduces f to 22 components. For the configuration vectors we consider three FS scenarios: `noFS`, `kindFS` and `aggFS`, yielding c vectors with, respectively, 23, 14 and 10 components. We then filter the PMLP dataset according to the FS scenario at hand. However, after this filtering, the dataset may contain points with the same (f, c) but different labels $p_{\texttt{ml}}(f, c)$. Thus, for each instance: a) we delete the dataset columns that FS left out; b) we perform `Pandas`'s `group_by` on the 22 columns chosen by FS, then c) compute the average $p_{\texttt{ml}}$ of each group and use this as the new label (at this point, rows with the same (f, c) have the same label); d) we remove the duplicate rows of the dataset by `Pandas drop_duplicates`, keeping only one row. Lastly, we select $\sim$11200 points for the PMLP.

4.2 PMLP experimental setup

The PMLP methodology of choice in this paper is SVR. Its advantages are: (a) the PMLP for training an SVR can be formulated as a convex Quadratic Program (QP), which can be solved efficiently; (b) even complicated and possibly nonlinear performance functions can be learned by using the "kernel trick" [**?**]; (c) the solution of the PMLP for SVR provides a closed-form algebraic expression

of the performance map $\bar{p}_{\mathcal{A}}$, which yields an easier formulation of $\mathsf{CSSP}(f, \theta)$. We use a Gaussian kernel during SVR training, which is the default choice in absence of any other meaningful prior [**?**]. We assess the prediction error of the predictor by Nested Cross Validation (NCV) [**?**,**?**]; furthermore, our training includes a phase for determining and saving the hyperparameters and the model coefficients of the SVR. These two tasks take approximately 4h in the `aggFS` and in the `kindFS` scenarios, and 5h in the `noFS` one. We use Python's `sklearn.model_selection.RandomizedSearchCV` for the inner loop of the NCV and a customized implementation for the outer loop, and `sklearn.svm.SVR` as the implementation of choice for the ML model.

A common issue in data-driven optimization is that using customary ML error metrics may not lead to good solutions of the optimization problem (see, for example, [**?**]). We tackled this issue by comparing the following metrics for the CV-based hyperparameter tuning phase, both computed on $p_{\texttt{m1}}$: the classical Mean Absolute Error $\mathsf{MAE} = \sum_{i \in S} |p_i - \bar{p}_i|$, where $p_i = p_{\texttt{m1}}(f_i, c_i)$ and $\bar{p}_i = \bar{p}_{\mathcal{A}}(f_i, c_i)$; the custom metric $\mathsf{cMAE}_\delta = \sum_{i \leq s} L_\delta(p_i, \bar{p}_i)$, where

$$L_\delta(p_i, \bar{p}_i) = \begin{cases} (\bar{p}_i - p_i) \cdot (1 + \frac{1}{1+exp(p_i - \bar{p}_i)}) & \text{if } p_i \leq \delta \text{ and } \bar{p}_i > p_i \\ (p_i - \bar{p}_i) \cdot (1 + \frac{1}{1+exp(\bar{p}_i - p_i)}) & \text{if } p_i \geq 1 - \delta \text{ and } \bar{p}_i < p_i \\ (p_i - \bar{p}_i) & \text{if } \delta \leq p_i \leq 1 - \delta \\ 0 & \text{otherwise} \end{cases}$$

and $\delta \in \{0.2, 0.3, 0.4\}$.

4.3 CSSP experimental setup

The choice of a Gaussian kernel in the SVR formulation makes the CSSP a MINLP with a nonconvex objective function $\bar{p}_{\mathcal{A}}$. More precisely, for a given instance with features $\bar{f}$, our CSSP is:

$$\min_{c \in \mathcal{C}_{\mathcal{A}}} \sum_{i=1}^{s} \alpha_i \exp\left(-\gamma \|(f_i, c_i) - (\bar{f}, c)\|_2^2\right) \tag{3}$$

where, for all $i \leq s$, (f_i, c_i) belong to the training set, α_i are the dual solutions of the SVR, γ is the scaling parameter of the Gaussian kernel, and $\mathcal{C}_{\mathcal{A}}$ is defined by mixed-integer linear programming constraints encoding the dependences/compatibility of the configurations. We use AMPL to formulate the CSSP, and the nonlinear solver Bonmin [**?**], manually configured (with settings `heuristic_dive_fractional yes`, `algorithm B-Hyb`, `heuristic_feasibility_pump yes`) and with a time limit of 60 seconds, to solve it; then we retrieve, for each instance f, the Bonmin solution c^*_{bm}. Since we have enumerated all possible configurations, we can also compute the "true" global optimum $c^*_{\mathsf{cssp}} = \arg\min\{\bar{p}_{\mathcal{A}}(f, c), c \in \mathcal{C}_{\mathcal{A}}\}$ for sake of comparison. In Table **??**, we report the percentage of cases where $c^*_{\mathsf{bm}} = c^*_{\mathsf{cssp}}$ ("%glob. mins") and, for all the instances where this is not true, the average distance between $\bar{p}_{\mathcal{A}}(c^*_{\mathsf{bm}})$ and $\bar{p}_{\mathcal{A}}(c^*_{\mathsf{cssp}})$, over all the instances of the considered set ("avg loc. mins"); we also report the average CSSP solution time ("CSSP time"). The `kindFS` and the `aggFS` scenarios

set	FS	%glob. mins	avg loc. mins	CSSP time
IS	noFS	83.69	**2.013E-02**	15.36
	kindFS	**87.70**	2.758E-02	10.73
	aggFS	84.49	3.122E-02	5.95
OS	noFS	85.32	**1.594E-02**	13.44
	kindFS	90.48	2.014E-02	12.23
	aggFS	**93.25**	1.957E-02	6.15

Table 1. Quality of Bonmin's solutions, w.r.t. $\bar{p}$

achieve better results, in terms of "%glob mins", than the noFS one. Furthermore, the local optima found by Bonmin are quite good ones: they are never larger than 3.2% (see "avg loc. mins") of c^*_{cssp}. The time that Bonmin takes to solve the CSSP is reduced from the noFS scenario to the aggFS one. This is due to the fact that the kindFS and the aggFS CSSP formulations have less variables than the noFS one, and so they are easier to solve. Bonmin needs, on average, less than 16 seconds to solve any CSSP; however, devising more efficient techniques to solve the CSSP (say, reformulations, decomposition, ...) might be necessary if our approach is scaled to considerably more algorithmic parameters.

4.4 Results

In order to assess the performance of the approach, we retrieve $p_{\mathsf{cpx}}(c^*_{\mathsf{bm}})$ and $p_{\mathsf{cpx}}(c_{\mathsf{cpx}})$ (CPLEX default configuration) from the filtered dataset, for every IS and OS instance. Tab. ?? and Tab. ?? show: the wins "%w" and the non-worsenings "%w+d", i.e., the percentage of instances such that $p_{\mathsf{cpx}}(c^*_{\mathsf{bm}})$ is $<$ or $\leq$ than $p_{\mathsf{cpx}}(c_{\mathsf{cpx}})$, by the first sixteen decimal digits of p_{cpx}, in scientific notation; the wins-over-nondraws "%w$_{\text{nond}}$", i.e., the percent wins over the instances such that $p_{\mathsf{cpx}}(c^*_{\mathsf{bm}}) \neq p_{\mathsf{cpx}}(c_{\mathsf{cpx}})$; the average $|p_{\mathsf{cpx}}(c_{\mathsf{cpx}}) - p_{\mathsf{cpx}}(c^*_{\mathsf{bm}})|$, over all the instances which score a win ("avg w") or a loss ("avg l"); the average $|p_{\mathsf{cpx}}(c^*_{\mathsf{bm}}) - \underline{p}_{\mathsf{cpx}}|$ over all the other instances, where $\underline{p}_{\mathsf{cpx}} = \min_{c \in \mathcal{C}_{\mathcal{A}}} p_{\mathsf{cpx}}(f, c)$ for a given f ("avg d").

The "%w", "%w+d" and "%w$_{\text{nond}}$" are higher on IS instances than on the OS ones: since the IS instances are used as the training set, it is not surprising that $\bar{p}_{\mathcal{A}}$ is less accurate at OS instances, which results in worse CSSP solutions. The fact that the "avg d" is always 0 implies that, whenever CPLEX — configured by c_{cpx} — manages to close the gap, our c^*_{bm} proves to be just as efficient. Percent draws ("%w+d" - "%w") are approximately 48-49% on IS instances and 46-47% on the OS ones. In the nondraws, our approach shows consistent gains with respect to c_{cpx}: "%w$_{\text{nond}}$" is approximately 94-95% on IS instances and 64-66% on the OS ones. From this we gather that $\bar{p}_{\mathcal{A}}$ provides an accurate approximation of p_{cpx}'s global minima, even at points outside the training set. In Tab. ??, the noFS scenario presents the worst results. Moreover, although the aggFS scenario achieves the highest percent wins on IS instances, the kindFS scenario dominates the others, by the highest "%w" on OS instances and by the best overall "%w$_{\text{nond}}$". However, the aggFS scenario achieves the largest average wins and suffers the smallest average losses. Tab. ?? shows that MAE-based CSSP formulations prevail on OS instances: using the MAE in the PMLP is conducive to very high generalization accuracy of $\bar{p}$. The second best choice is

set	FS	%w	%w+d	$\%w_{nond}$	avg d	avg w	avg l
IS	noFS	47.06	96.12	92.39	0.000E+00	3.400E-06	**6.493E-01**
	kindFS	48.91	98.02	**96.12**	0.000E+00	3.415E-06	8.895E-01
	aggFS	**48.98**	97.79	95.70	0.000E+00	**3.438E-06**	8.530E-01
OS	noFS	33.73	83.73	67.50	0.000E+00	3.148E-06	7.042E-01
	kindFS	**36.64**	82.61	**68.06**	0.000E+00	3.051E-06	5.385E-01
	aggFS	33.86	77.05	59.78	0.000E+00	**3.335E-06**	**4.486E-01**

set	metric	%w	%w+d	$\%w_{nond}$	avg d	avg w	avg l
IS	cMAE_2	47.33	97.09	94.80	0.000E+00	3.434E-06	7.523E-01
	cMAE_3	**48.46**	97.53	95.19	0.000E+00	3.390E-06	9.496E-01
	cMAE_4	**48.46**	97.59	**95.27**	0.000E+00	**3.443E-06**	7.656E-01
	MAE	47.86	96.73	93.62	0.000E+00	3.401E-06	**6.895E-01**
OS	cMAE_2	34.74	79.72	63.34	0.000E+00	3.205E-06	5.710E-01
	cMAE_3	33.07	80.51	63.33	0.000E+00	3.320E-06	**5.392E-01**
	cMAE_4	35.63	81.39	65.99	0.000E+00	**3.274E-06**	5.869E-01
	MAE	**36.16**	85.36	**71.78**	0.000E+00	2.982E-06	5.812E-01

Table 2. Pipeline quality w.r.t. $p_{\mathsf{cpx}}(c^*_{\mathsf{bm}})$, by FS scenario and IS/OS set

Table 3. Pipeline quality w.r.t. $p_{\mathsf{cpx}}(c^*_{\mathsf{bm}})$, by PMLP metric and IS/OS set

the cmae_4, which attains the best performances on IS instances and also scores the largest overall wins ("avg w").

5 Conclusions

All in all, the results show that our approach is promising, in that it provides configurations that are better than those provided by the heuristics inside CPLEX. The results also show that a number of important details have to be properly accounted for before the approach can deliver good performances. In particular, it is interesting that, in some cases, the custom cMAE error metric has higher generalization capabilities than the classical MAE, which nonetheless outperforms our custom metric on OS instances. Since the choice of the PMLP error metric determines the quality of the CSSP solutions, this can be expected. From the results reported in Tab. **??**, we gather that performing FS in the PMLP can also improve the quality of the CSSP solutions. Yet, this indicates that applying ML techniques to the ACP requires taking into account the specific aspects of the problem.