A Classifier to Decide on the Linearization of Mixed-Integer Quadratic Problems in CPLEX*

Pierre Bonami
CPLEX Optimization, IBM Spain
pierre.bonami@es.ibm.com

Andrea Lodi
Polytechnique Montréal
andrea.lodi@polymtl.ca

Giulia Zarpellon
Polytechnique Montréal
giulia.zarpellon@polymtl.ca

Abstract

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

1 Introduction

While mathematical optimization intrinsically lies at the core of machine learning (ML) methods, recent years have seen a rise in the application of learned approaches to discrete optimization settings [2]. In particular, a successful paradigm has been identified in using ML within Mixed-Integer Programming (MIP) algorithmic frameworks as a way of complementing the capabilities of a solver and providing indications about structural decisions for which we lack in-depth understanding.
We position our work in this recent yet very fruitful research area, and consider Mixed-Integer Quadratic Programming problems (MIQPs). Despite the fact that modern MIQP solvers – and among those IBM ILOG CPLEX [9], our solver of choice – have been able to solve MIQPs for several years (see, e.g., [5]), the theoretical and computational implications of the employed resolution techniques are not fully grasped yet. We are interested in understanding whether to linearize the quadratic part of a convex MIQP, a decision that substantially conditions the downstream resolution algorithms operated by a solver. Currently, CPLEX users can utilize a switch parameter to specify whether a linearization step should take place during preprocessing, but it is not clear when this switch should be turned on or off in order to benefit the resolution process: when one considers a wide variety of problems the decision about whether to linearize is not clear cut.
Our goal in this paper is to use ML statistical tools to decide whether to linearize a convex MIQP or not. We make the empirical conjecture that some of the reasons leading to an algorithmic

*This work is an extension of [6].

discrimination between the linearization approach (L, in short) and the non-linearization one (NL) might be linked to the formulation characteristics and the early stages of the optimization of a MIQP problem, and could hence be detected by a learning algorithm if enough relevant information was provided as input. The idea that perhaps MIQPs should be solved in a more flexible and adapted way was first suggested in [12], and naturally calls for a predictive machinery. In this sense, the question *linearize vs. not linearize* qualifies as a good quest for ML techniques, and it is naturally framed in a classification setting. We began to explore such classification approach in [6]: the developed framework took care of building a synthetic dataset, designing features and labels, and conducting preliminary learning experiments. We also defined new metrics to assess the quality of the prediction from the optimization standpoint, i.e., in terms of runtimes. Results were satisfactory, but limited by the fact that only artificial MIQPs were used. Moreover, the offline learning phase was not integrated in the solver.

1.1 Contributing a methodology

In the present work, we extend what was done in [6] along different directions, aiming at a tighter combination of the learning and the optimization perspectives. We resume from what we identified in [6] as future research plans and

- enlarge our dataset to include non-synthetic benchmark instances: we add to our pool of problems MIQPs from NEOS [10, 11, 14] submissions and CPLEX internal testbed;
- extend the feature design and feature selection process to achieve more detailed representations of MIQPs, of which we carry out a careful analysis;
- perform new learning experiments and explore different ways to incorporate optimization knowledge in the learning pipeline.

Finally, we implement our predictive framework in the solver ecosystem: as a practical outcome, a learned classifier is deployed in CPLEX 12.10.0. These contributions allow us to establish the first example of an end-to-end integration of ML tools into a leading commercial solver – from the definition of an appropriate ML task responding to the algorithmic question of whether to linearize a MIQP, until the final deployment of the obtained prediction function within a complex solver environment. We believe that the methodology we designed over time, starting from the early attempt [6], could be applied to a variety of other heuristic tasks in the solver, and will serve as a reference in an area that is rapidly evolving and gaining attention. In this sense, the present work ultimately contributes a methodological process for the combination of ML and MIP technology: we share the questions that guided us in the development, the decisions and turns we had to take and the motivations behind them.

The paper content is shaped upon our methodological steps, which are outlined in Figure 1. We start by examining the MIQP algorithmic framework of CPLEX (Section 2), in order to identify and properly delimit our learning question (e.g., in terms of which MIQPs and algorithms are involved). The next step is building a dataset (Section 3). Targets capture the essence of a learning question, so their definition is of utmost importance, and we discuss two valid labeling procedures for discriminating between the L and NL methods. We approach feature design with questions like: what factors could be important for our decision? which traits of MIQPs might play a role in the algorithms we are trying to compare?, and address the need to gather (and generate) MIQP instances for data collection.

We follow ML best practices when defining learning experiments (Section 4), but we complement them with context-specific measures to evaluate the classifier performance in the solver. In fact,

I. **Understanding the algorithmic framework**
  - Identify and delimit the learning question

II. **Building a dataset**
  - Target definition via labeling procedure
  - Feature design to represent the decision
  - Gathering data: instances and analysis

III. **Learning experiments**
    – Best practices and context-relevant evaluation metrics
  - Baseline results
    – Assessing feature importance
  - Framework revision: necessary adjustments, feature selection
  - Adding domain-specific priors to the learning phase

IV. **Implementing predictions in the solver ecosystem**
    – Fine-tuned workflow

Figure 1: ML in MIP technology: methodological steps.

standard ML indicators cannot provide information on the impact of misclassification in terms of the metric that we use to compute targets. Baseline results serve us to verify the soundness of our approach and get an idea about the importance of the represented features, but the initial framework needs to be adjusted to be embedded in the solver. We ask ourselves what is ultimately viable and what changes are necessary to incorporate predictions in CPLEX: answers to such questions lead us to a substantial feature selection phase. To further condition predictions towards our true performance goal, we introduce domain-specific priors in the learning phase, and eventually recover information that was previously sacrificed. The experimental phase proceeds far from linearly, and we iterate step III (Figure 1) to attain satisfactory results. Finally, we discuss in Section 5 the practical implementation of a predictor in the CPLEX optimization pipeline, the required fine-tuning and the achieved outcome, before some concluding remarks in Section 6.

2 The MIQP algorithmic framework in CPLEX

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

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

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

When integrality requirements are dropped from (1), one obtains the (continuous) Quadratic Programming (QP) relaxation of the problem. If the matrix Q is positive semi-definite ($Q \succeq 0$), the quadratic form to minimize is convex and the corresponding QP can be solved in polynomial time; in this case, the QP relaxation is thus called *convex*. In the present work, we restrict ourselves to MIQPs whose Q matrix is positive semi-definite or can be made positive semi-definite by simple transformations, i.e., we only consider MIQPs that are usually regarded as "convex" by state-of-the-art

MIQP solvers. Two simple transformations that can be applied to repair the indefiniteness of Q are the following.

Linearization of products involving binary variables Consider a binary variable x_j of (1), i.e., $l_j = 0$, $u_j = 1$ and $j \in I$. The product of x_j with any other bounded variable x_i satisfies the following linear inequalities [20]:

$$\max \left\{ \begin{array}{c} u_i x_j + x_i - u_i \\ l_i x_j \end{array} \right\} \leq x_i x_j \leq \min \left\{ \begin{array}{c} l_i x_j + x_i - l_i \\ u_i x_j \end{array} \right\}. \tag{2}$$

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

Perturbation of the diagonal of Q for binary variables Again using the fact that for a binary variable $x_j^2 = x_j$, one has $x^T Q x + \sum_{j \in B} \rho_j \left(x_j^2 - x_j \right) = x^T Q x$, where $B \subseteq I$ denotes binary variables. The principal minor of Q corresponding to variables in B can thus be made positive semi-definite. In particular, if all non-zero products in Q involve at least one binary variable, Q can always be perturbed so that the resulting QP relaxation is convex. Note that the choice of an appropriate ρ is a non-trivial step. A simple way to ensure that the perturbed quadratic form $x^T Q x + \sum_{j \in B} \rho_j x_j^2$ has no negative eigenvalue is to directly use Q eigenvalues, though more advanced techniques leverage semi-definite programming and the linear constraints of (1) to produce tight QP relaxations [1, 3].

Leading solvers for MIQPs can perform either of the two operations above (linearize or perturb) at the beginning of the optimization, in a preprocessing phase. If the resulting problem has a convex QP relaxation it will be solved as a *convex MIQP*; otherwise, solving the QP relaxation itself is an $\mathcal{NP}$-hard problem [22] and more involved techniques are required. In this paper we only consider the former case. The state of the art for solving convex MIQPs usually employs computationally efficient algorithms for solving QP relaxations; in CPLEX, a simplex-based algorithm is preferred for its good restart properties. For the rest, the technology is similar to the one used for Mixed-Integer Linear Programs: a branch-and-bound tree search, augmented with cutting planes techniques, and heuristic procedures to obtain good feasible solutions [17]. Note that both approaches can also be applied when initially $Q \succeq 0$ – and in practice they are. In particular, the linearization step could reformulate an already convex MIQP into a Mixed-Integer Linear Program, thus deciding which resolution method and technology are applied to solve the problem.

2.1 The linearization option

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

As reported in [5], the linearization approach does not dominate in theory the non-linearization one. In CPLEX internal experiments, though, linearizing appeared to be superior on average to tackle convex MIQPs, and became the default method. Since version 12.6.0, CPLEX provides to users the possibility to switch the linearization mechanism on or off through the preprocessing parameter `qtolin`[1], whose automatic value corresponds to always linearize. But the linearization option is not always beneficial, as was reported in [12]. The following example shows that a range of situations can occur.

Example 2.1. We generate MIQP instances (see Section 3) of varying size and structural properties, and run CPLEX with both `qtolin` on and off with five different random seeds and a time limit of 2h. The following table reports for five problems the shifted geometric means of runtimes for the L and NL strategies, together with the problems' number of variables and constraints (n, m), the density of the Q matrix, and the percentage of "hard" eigenvalues of Q (i.e., those making the starting Q indefinite).

Table 1: Structural parameters and average runtimes for five synthetic MIQP instances.

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

Clearly, the initial convexity of Q itself does not decide which method between L and NL is the best suited to solve a MIQP, and Q density does not define the best option either. Possibly, a combination of many factors together could parameterize the best solving mode. Note that choosing the correct strategy for problems A and E appears critical, in the sense that a wrong decision could result in the problem not being solved within the time limit. In contrast, for problems B and C the performance gap of L and NL is less pronounced, while for D the two methods are practically equivalent.

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

3 Building a dataset

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

3.1 Labeling procedure

Given our question L vs. NL, we need to provide for each MIQP the answer corresponding to the better performing approach. We identify three possible scenarios, and therefore assign one among

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

Algorithm 1 Checks on MIQP runs

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

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

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

Consistency check

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

  return `False` ▷ discard the run because inconsistency was found

end if

Solvability check

`solved` = [`optimal`, `infeasible`] ▷ Statuses corresponding to solved runs

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

  return `False` ▷ discard the run because not solved within timelimit

end if

return `True`

three categorical labels: L (*linearize*, i.e., `qtolin` on), NL (*not linearize*, i.e., `qtolin` off), and T (*tie*), when L and NL methods are comparable in terms of performance. Tracking tie cases provides a way of distinguishing between critical and non-critical problems, and can be helpful when evaluating the learned predictions.

To deal with the solver's performance variability [18], each instance is run in both `qtolin` modes with five different random seeds. We enforce a timelimit of 2h for each run, and collect data on final upper and lower bounds, resolution times and solver's solution statuses. We implement two checks to remove troublesome runs:

- Consistency check: on each seed, we compare best primal and dual bounds achieved by methods L and NL; when an inconsistency is found, the run on that seed is discarded;
- Solvability check: the run on a seed is discarded if neither L nor NL were able to solve the problem to optimality.

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

In addition, we define a binary labeling scheme BinLabel, reported in Algorithm 3. Unlike the multi-class procedure, BinLabel does not take into account consistent seed wins. Instead, it directly compares the shifted geometric means of running times for L and NL, across the seeds passing the checks. Eventually, ties are broken using shifted geometric means of the number of nodes. The comparison of computing times in BinLabel does not use a 10% threshold, but the resulting scheme is

Algorithm 2 Labeling procedure – MultiLabel case

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

Seed wins

$wins_{\mathsf{L}} := 0, wins_{\mathsf{NL}} := 0$
`solved` = [`optimal`, `infeasible`] ▷ Statuses corresponding to solved runs
for seed s in passed seeds **do**
  if ($status_{s,\mathsf{L}} \in$ `solved`) and ($status_{s,\mathsf{NL}} \in$ `solved`) **then**
    if $t_{s,\mathsf{L}} < (1-p)t_{s,\mathsf{NL}}$ **then** ▷ L significantly better than NL on s
      $wins_{\mathsf{L}} \leftarrow wins_{\mathsf{L}} + 1$
    else if $t_{s,\mathsf{NL}} < (1-p)t_{s,\mathsf{L}}$ **then** ▷ NL significantly better than L on s
      $wins_{\mathsf{NL}} \leftarrow wins_{\mathsf{NL}} + 1$
    end if
  end if
  if ($status_{s,\mathsf{L}} \in$ `solved`) and ($status_{s,\mathsf{NL}} \notin$ `solved`) **then**
    $wins_{\mathsf{L}} \leftarrow wins_{\mathsf{L}} + 1$
  else if ($status_{s,\mathsf{L}} \notin$ `solved`) and ($status_{s,\mathsf{NL}} \in$ `solved`) **then**
    $wins_{\mathsf{NL}} \leftarrow wins_{\mathsf{NL}} + 1$
  end if
end for

Winner label assignment

if $wins_{\mathsf{L}} \geq wins_{\mathsf{NL}} + \Delta$ **then return** label L
else if $wins_{\mathsf{NL}} \geq wins_{\mathsf{L}} + \Delta$ **then return** label NL
else return label T
end if

Algorithm 3 Labeling procedure – BinLabel case

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

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

Time and nodes comparison

if $Time_{\mathsf{L}} < Time_{\mathsf{NL}}$ **then return** label L
else if $Time_{\mathsf{NL}} < Time_{\mathsf{L}}$ **then return** label NL
else
  if $Nodes_{\mathsf{L}} < Nodes_{\mathsf{NL}}$ **then return** label L
  else if $Nodes_{\mathsf{NL}} < Nodes_{\mathsf{L}}$ **then return** label NL
  else return `None`
  end if
end if

nevertheless consistent with the MultiLabel one: L and NL labels assigned in the multi-class procedure remain the same in the binary one. In other words, one can interpret BinLabel as a way of turning T samples obtained with MultiLabel into L and NL cases.

3.2 Feature design

A raw formulation like (1) cannot be fed directly as input to a learning algorithm, so we need to represent a MIQP via a vector of numerical features $\mathbf{x} \in \mathbb{R}^d$, which should condense what we suspect are the important pieces of information leading to an algorithmic discrimination between L and NL. We describe a MIQP instance in its mathematical, optimization and computational properties, by means of a set of 60 hand-crafted features. For feature design, we reinterpret few ideas from the recent works [16] and [15]. We mostly capture *static* information from the initial formulation, but given the impact of the linearization and perturbation steps on the quality of the root dual bound [1] (and hence on the success of the subsequent optimization process), we also extract data from the *preprocessing* phase and the resolution of the *root node relaxation*, for both L and NL. Features are defined in such a way to be comparable across a variety of instances, as they should express common characteristics of MIQPs. With respect to our early work [6], the feature set has been revised and extended, with the goal of better capturing the composition of matrix Q and the changes induced by preprocessing.

Static features Properties of a MIQP that can be read from the formulation (1) are basic information about the size of the problem (number of variables and constraints) and the proportions of variables of each type (binary, general integer and continuous). The composition of the symmetric matrix Q can be detailed by inspecting the presence of different types of non-zero bilinear products, in and out of the main diagonal, and in particular the appearance of non-linearizable terms. Additionally, we examine spectral properties of Q such as rank and proportions of zero and "hard" eigenvalues. Connectivity degrees of variables appearing in x^TQx are also tracked, and we compute proxies for potential increases in variables and constraints sizes after linearization. We record the composition and density of the linear term c of the objective function. For constraints, we inspect variables' involvement (per type) and the density of matrix A.

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

Root node features To measure the performance difference between L and NL in solving their respective relaxations, we collect and compare runtimes and dual bounds achieved after the root node resolutions.
In total, we collect 44 static, 11 preprocessing and 5 root node features; we report them in Table 9 in the Appendix. At this stage, features are still extracted offline: we compute static traits with the CPLEX Python API after reading MIQP instances. Necessary information from preprocessing and the root node resolutions of L and NL are gathered during the runs of the labeling procedure (Section 3.1); dual bounds and runtimes at the root node are aggregated using arithmetic and shifted geometric means, respectively, across the runs that passed the checks of Algorithm 1.

Table 2: Dataset composition in terms of labels, for both labeling schemes. Percentages refer to the corresponding row counts (#).

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

3.3 Instances

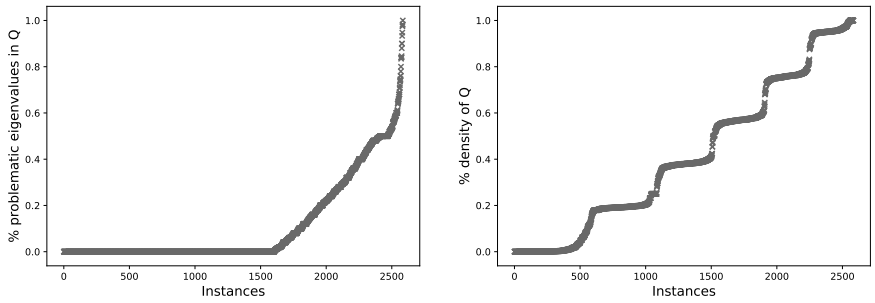
We aim to compile a dataset of MIQPs that is heterogeneous and relevant for the L vs. NL question, and representative of the variety of cases that can occur, as we saw in Example 2.1. Driven by the need of more problems than what available libraries offer, we first create synthetic MIQP instances. The generation procedure takes into account different structural parameters (such as size, density and spectrum of Q) and multiple types of constraints to obtain heterogeneous MIQPs. We refer to the resulting dataset of 2640 MIQPs as `setD`, and to [6] for more details on data generation.
Examining the problems that were generated for [6], we observed the presence of instances with high density due to a dense encoding and near-to-zero coefficients q_{ij}. We hence decided to apply a numerical correction to the dataset, enforcing sparsity of Q matrices. On the one hand, such correction disrupted the spectral properties of some unstable instances, which could now be read by the solver as general nonconvex ones, and consequently rejected.[2] On the other hand, we deem the corrected instances to be more stable and meaningful than their original versions. Instances of `setD` have been contributed to the MINOA open-source benchmark library [21].
For this work, we enlarge our MIQP dataset to include non-synthetic benchmark instances from NEOS submissions and problems of CPLEX internal testbed:

- `neos` contains 945 MIQPs that were submitted to the NEOS server with CPLEX as the specified solver. Instances have been collected from submissions between April 2015 and January 2018 and cleaned for duplicates;
- `miqp` contains 522 problems that constitute the CPLEX internal MIQP dataset. Differently from `setD`, `miqp` is dominated by the presence of very structured combinatorial MIQPs, like Max-Cut and Quadratic Assignment Problems. Note that instances from the literature (e.g., QPLIB ones [13]) are also included in this set.

Altogether, `setD`, `neos` and `miqp` amount to 4107 MIQP instances, which we run in both `qtolin` modes, for five random seeds, on a cluster of identical 12 core Intel Xeon CPU E5430 machines running at 2.66 GHz and equipped with 24 GB of memory, with CPLEX version 12.8.0. After performing Consistency and Solvability checks, 2585 problems remain ($1821, 480, 284$ from `setD`, `neos` and `miqp`, respectively). We then compute labels with both schemes MultiLabel and BinLabel, and extract features as described in Section 3.2. The composition of the dataset in terms of labels is reported in Table 2; note that a proper BinLabel could not be assigned in 935 cases due to ties both in runtimes and number of nodes[3], so that only 1650 problems are available in the binary setting.

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

[3]This happens because we set all runtimes smaller than or equal to 0.1 seconds to be precisely 0.1.



(a) (b)

Figure 2: (a) Fraction of problematic (hard) eigenvalues of Q, and (b) density of Q in the full dataset (2585 instances).

While the MultiLabel scheme produces almost 55% of tie cases, BinLabel yields a 70-30% repartition between L and NL, respectively. As shown in Figure 2a and 2b, the proportion of problematic eigenvalues of matrix Q, as well as its density, span the entire [0,1] range in the full dataset.

4 Learning experiments

We perform learning experiments on the entire dataset with multi-class labels $\{\mathsf{L}, \mathsf{NL}, \mathsf{T}\}$ given by MultiLabel (2585 points), as well as on the binary subset of samples with targets defined by the BinLabel procedure (1650 points).
As classification models, we test Logistic Regression (`LogReg`), Support Vector Machine (`SVM`) with RBF kernel [8], a single Decision Tree (`Tree`) and Random Forests (`RF`) [7]. We specify a grid of values for the main hyper-parameters of each model, in order to search the best combinations. All models can perform both multi-class and binary classification, and are compared with a dummy classifier (`dum`) following a stratified strategy (i.e., generating predictions according to the class distribution of the training set).
For each of our learning experiments, we randomly split the available data into a training and a test sets using a 75-25% ratio. We perform a training phase with 5-fold cross validation to grid-search models' hyper-parameters, and a test phase on the neutral test set. When splitting the data and defining folds, general proportions of labels are maintained in each subset; features are standardized with respect to each subset, by removing the mean and scaling to unit variance. Each type of experiment is repeated for five different random seeds: within the learning pipeline, randomization mostly affects the determination of the train/test splits, but can also impact the definition of some predictive models (e.g., `RF`). Practically, learning experiments are implemented in Python 3.5 with Scikit-learn 0.20.0 [23], and run on a dual Intel(R) Xeon(R) Gold 6142 CPU @ 2.60GHz, equipped with 512GB of RAM.

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

predictions over a (test) set of size K as

$$\text{accuracy}(y, \hat{y}) = \frac{1}{K} \sum_{k=1}^{K} \mathbb{1}_{\{y^k = \hat{y}^k\}}. \tag{3}$$

We also report f1-scores, i.e., harmonic means of precision and recall for the predictions (see [4] for details). More generally, a weight w^k can be associated to each sample k, to get a *weighted* accuracy score

$$w\text{-accuracy}(y, \hat{y}, w) = \frac{1}{\sum_k w^k} \sum_{k=1}^{K} w^k \cdot \mathbb{1}_{\{y^k = \hat{y}^k\}}. \tag{4}$$

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

$$\text{b-accuracy}(y, \hat{y}) = w\text{-accuracy}(y, \hat{y}, w_{class}). \tag{5}$$

However, from the perspective of practically solving MIQPs, misclassifying critical problems has more severe effects than predicting the wrong method for an instance in which L and NL show instead comparable performances. As we saw in Example 2.1, misclassifications are neither all equally important nor bad. In order for our measurements to reflect the quality of the predictions from the solver's standpoint, we introduce a notion of sample weights linked to the runtimes of L and NL. A natural way of measuring how critical a MIQP problem is – i.e., how different the methods perform, and hence how important it is to classify the sample correctly – is that of considering the shifted geometric mean of the runtimes difference, as in

$$w_{time} := sgmean(|t_{s,\mathsf{L}} - t_{s,\mathsf{NL}}| : s \in \text{seeds}). \tag{6}$$

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

$$\text{t-accuracy}(y, \hat{y}) = w\text{-accuracy}(y, \hat{y}, w_{time}). \tag{7}$$

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

$$\sigma_{clf} := \sum_{k=1}^{K} t^k_{clf}, \tag{8}$$

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

Table 3: Baseline results for the two data setting; 60 `Initial` features are used for learning. Reported values are averages across five experiments of the same type, on different seeds.

(a) MultiLabel, `Initial` features

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

(b) BinLabel, `Initial` features

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

4.1 Baseline results

As a baseline experiment, we train classifiers on the initial set of 60 features (`Initial`), in both multi-class and binary settings. Results reported in Table 3 are averages of the scores across five tries. With respect to traditional classification measures, all classifiers are exhibiting good performance, with `RF` and `SVM` usually being the best performing models. Scores accounting for runtimes in their definition (i.e., t-accuracy, $\sigma_{clf}/\sigma_{\texttt{target}}$, and $\sigma_{clf}/\sigma_{\texttt{def}}$) appear consistent with the classification ones. In particular, `RF` yields at least a 14% improvement on `def` in both settings: `def` uses 16% or more time than `RF` to solve test instances. Classification scores in the multi-class configuration are generally higher than those in the binary one. An inspection of the confusion matrices allows to assess that classifiers in the multi-class setting are in fact very good at correctly classifying T cases, with errors mostly happening when distinguishing between L and NL. Given that the BinLabel scheme in fact transforms some tie cases into L and NL samples, the classification of T which was so accurate in the MultiLabel setting translates into less clear-cut separation in the binary one. Nonetheless, despite misclassification happening more frequently in the BinLabel setting, t-accuracy and ratios of prospective runtimes remain high for `RF` and `SVM`. Again, misclassifications do not have all the same impact in terms of solver performance, and runtime-based metrics show that classifiers are still able to predict correctly on many critical binary samples. For these reasons, we decide to focus our subsequent experiments in the binary setting only.

Feature importance To get a sense of the importance of each feature in the prediction, we analyze the importances scores of the trained `RF` models. Such scores, computed by Scikit-learn, consist of nonnegative scalar values summing up to 1 (among all features), and represent the mean decrease in impurity [19] for each feature. Simply put, the predictive power of an attribute is

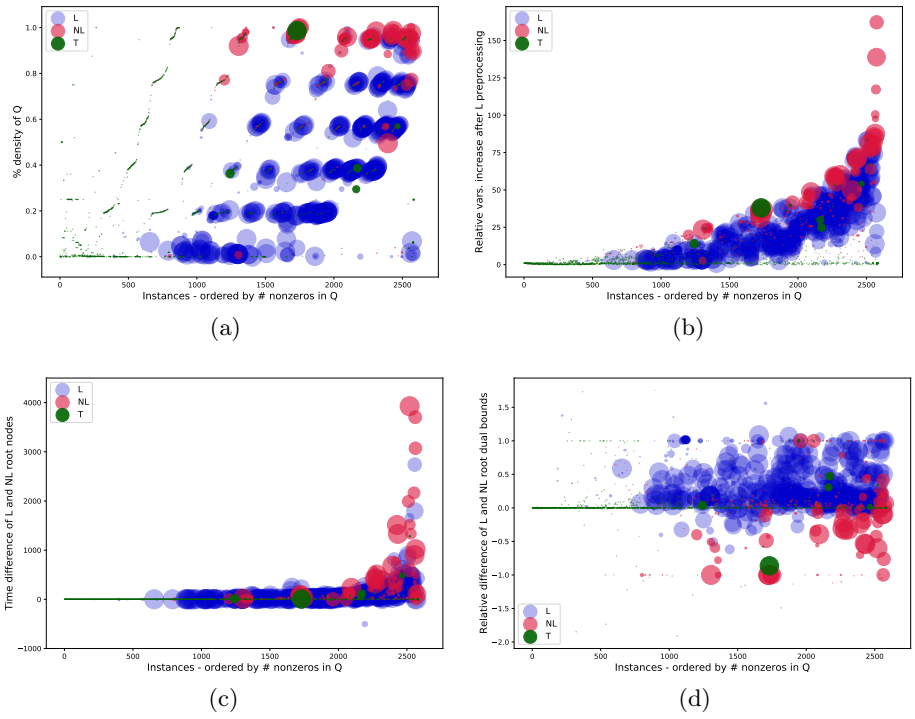


Figure 3: Four relevant features in the full dataset: (a) density of Q, (b) relative increase in number of variables after L preprocessing, (c) root node runtimes difference between L and NL, (d) root node bounds difference between L and NL. Colors match MultiLabel targets, while bubble size is proportional to weights w_{time}. In all plots, instances are ordered by the number of non-zeros in Q.

quantified in terms of the depths in the decision tree at which the attribute is used to create a node split, following the rationale that attributes used at the top practically affect more samples. We average the features' scores across the five RF models trained in both data setting, and consider the 10 top-ranked features, which we report in Table 10 in the Appendix. For MultiLabel, top features are mostly from preprocessing and the root node resolution; this suggests that T cases can be well detected thanks to this type of non-static information. In the binary configuration, instead, a common subset of features starts to emerge: together with few non-static features, attributes describing the composition of matrix Q appear. In particular, measures of density and of the presence of binary variables gain relevance. In Figure 3 we plot four relevant features across the multi-class dataset. The density of Q (Figure 3a) and the relative dual bound difference between L and NL (Figure 3d) clearly help in discriminating between classes. Together with the relative increase in number of variables after L preprocessing (Figure 3b), they also reflect the trade-off between size and strength of the (re-)formulation, which is crucial for a successful resolution of MIQPs [1] .

4.2 Feature selection

Up to now we performed data collection and learning experiments offline, tracing over the outline of what had been done in [6]. However, the ultimate goal of the present work is that of tightly integrating a predictive tool with the MIQP solver, a task that requires rethinking our initial framework and adjustments of various kind. Mainly, instead of data being collected once and for all, feature extraction will need to be performed online, when a MIQP instance is presented to the

Table 4: Description of features in the `Selected` subset (21).

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

solver. More generally, one has to understand when the prediction should take place, with respect to the resolution pipeline and the solver's various functionalities – a consideration that, in turn, affects which type of input can be available for the predictor model. Moreover, not all the hand-crafted features prove to be useful for good classification: in fact, the presence of irrelevant features in the input may induce over-fitting, besides entailing extra computational cost. One generally needs to compromise between the predictive power of some features and the possibility of efficiently computing them in the solver: attributes related to root node information are certainly useful for classification but expensive to get, as they would require to solve the root node twice. Some static features involving a spectral decomposition of Q are also not viable in an online procedure.
Practically, these considerations altogether motivate the revision of the hand-crafted feature set. We drop features that are not accessible for an online solver computation; in particular, we remove from the initial 60 traits:

- most of the features regarding the spectrum of the Q matrix, only keeping information on the proportion of hard eigenvalues and zero ones (i.e., measures relative to Q rank);
- features gathered from root nodes resolution for both L and NL, and those from the preprocessing step of NL;
- features prone to numerically ill behaviors (e.g., those involving comparisons of A, b and c coefficients), in order to avoid scaling issues in the learning phase and remove dependencies from each instance's parameters.

Table 5: Results for the binary setting using the 21 `Selected` features. Reported values are averages across five experiments of the same type, on different seeds.

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

After these reductions, we end up with a set of 35 features. Note that the only non-static features kept are those extracted from the `L` preprocessing step, which is more expensive than the `NL` one, but appeared more useful for predictions in the baseline experiments. We then proceed with a phase of further feature selection. Feature selection is an inherently iterative phase in the learning pipeline, and we try various procedures – from filtering based on a single SVM model or Decision Tree, to cross-validating the best subset using ensemble methods. We also examine the performance of different feature subsets with respect to the regression task that will be included in the learning pipeline (see Section 4.4). At the end, we decide to keep a subset of 21 features (17 static and 4 from `L` preprocessing) that are consistently identified throughout the experiments as relevant and leading to satisfactory classification performances. We describe this final subset of attributes in Table 4, and refer to it as `Selected`.
We run classification experiments in the binary setting; results are reported in Table 5. All scores appear decreased with respect to the baseline (cf. Table 3); in particular, performance in terms of prospective runtimes is generally reduced. Relevant features consolidate in the binary configuration, consistent with what already observed in the baseline experiments: with features describing the initial Q composition (particularly in terms of binary variables appearance), traits on the effects of linearization (e.g., with respect to problem size and A's density) are regularly in the top-10 (cf. Table 10).

4.3 Using runtime weights

After a necessary reduction in input features, classification and runtime-related scores dropped considerably. In this and the next sections, we introduce some changes in the learning process to condition predictions towards our true performance goal and thus improve the optimization performance of the classifiers. Improving the predictions from the optimization point of view concretely means making the classifiers more attentive to critical data-points. A very natural idea to incorporate the knowledge of whether a sample is critical is to use w_{time} as defined in (6) as *sample weights*, i.e., to work with a weighted dataset. In SVM models, for example, the use of weights on the points has the effect of re-scaling the penalty parameter, so that during training the classifier will be emphasised to get high-weight points correctly. In a single Decision Tree, instead, weights would modify the classes' probabilities in the identified split regions of the feature space.
Another possible way of introducing a prior about critical instances is by defining a custom loss function designed to penalize misclassification proportionally with the criticality of samples. In this respect, we define

$$\texttt{WTarLoss}(y, \hat{y}) := \frac{1}{\sigma_{\texttt{target}}} \sum_{k=1}^{K} w_{time}^{k} \cdot \mathbb{1}_{\{y_k \neq \hat{y}_k\}}, \tag{9}$$

Table 6: Results for the binary setting using the `Selected` feature subset (21), with sample weights w_{time} and custom cross-validation scoring function `WTarLoss`. Reported values are averages across five experiments of the same type, on different seeds.

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

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

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

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

a weighted loss with respect to `target` runtimes. For a misclassified sample k ($y_k \neq \hat{y}_k$), the weight w^k_{time} is a proxy for the difference between the prospective runtime of the classifier and the `target` one, i.e., of $t^k_{clf} - t^k_{\texttt{target}}$. We try `WTarLoss` as scoring function for cross-validation, so that during training the best combinations of classifiers' hyper-parameters will be chosen based on this score (the lower the better).

Results in both the sample weights setting and with the use of `WTarLoss` are reported in Table 6. Overall, it appears clear that incorporating some prior knowledge on which instances are critical via weights substantially helps to improve the classifiers prospective optimization performance. Measures of t-accuracy and runtimes ratios strengthen in both setups, with sample weights especially boosting `SVM` (cf. Table 6a). Classification measures are comparable between settings, and with respect to previous results in which no weight information were used (cf. Table 5). Note that the use of sample weights leads to lower b-accuracy scores, i.e., there is a marked imbalance in terms of which class is correctly predicted. In this setup, confusion matrices reveal that L is the predicted label for a higher number of samples, and most misclassifications happen in the form of a NL wrongly predicted as L. This might be linked to the fact that values of w_{time} show different distributions when restricted to L and NL samples; Table 7 presents descriptive statistics for weights in BinLabel data. While $w_{time} \in [0, 7200)$ hits the same min and max values for both classes, mean and 75% percentile values indicate that weights are higher for L samples. In other words, there is more to lose (on average, in our data) when misclassifying L for NL than vice versa.

Table 7: Statistics for w_{time} with respect to classes, in the BinLabel data. We report: count, mean, standard deviation, min, max and percentiles values.

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

4.4 Regression of root bounds information

Among the factors affecting the final runtime of a MIQP, an important one clearly is the quality of the dual bound reached at the root node, i.e., the strength of the initial problem relaxation. As we already noticed, features comparing root node information of L and NL methods were deemed very useful for correct predictions (cf. Table 10 and Figure 3d). We try to incorporate dual bounds information without solving the root node twice, by approximating a root feature via learned regression. In particular, we select `root_RelSignRDBDiff`, i.e., the relative signed difference of root dual bounds in L and NL, and use it as target to train a Support Vector Regression model (`SVR`) with a nonlinear RBF kernel. For the minimization case (dual bounds are lower bounds),

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

the metric being positive when L's bound is better, negative otherwise. Recall that bounds lb_{L} and lb_{NL} are arithmetic means of those benchmarked during the labeling procedure. In fact, this feature partially aligns with the binary labeling: when linearizing is the best choice, that will be reflected in the bound quality 91% of the times; when NL is the target, instead, the NL bound is actually better than the L one only 23.6% of the times (cf. also Figure 3d).
Practically, we need to allocate part of our data to train the `SVR`. In our experimental pipeline we now first perform a 30-70% split for training and testing the regression; we restrict classification to the `SVR` test set only, i.e., we further divide the 70% portion into a 75-25% split for training and testing classification models. The predictions of the trained `SVR` are added to the 21 input features of the `Selected` subset, and used for classification. Note that the entire classification phase only relies on *predicted* values of `root_RelSignRDBDiff`, never true ones.
The average mean square error of `SVR` across five experiments is 0.1012. Classification-wise, results improve when the regressed root information is exploited. Table 8 reports scores for both kind of weights integration. If classification metrics are comparable with the previous cases, the use of `SVR` is particularly helpful to improve runtime-related scores. In the sample weight case (Table 8a) models improve their performance, the only exception being `SVM`. When `WTarLoss` is used instead (Table 8b), `SVM` is the best performing classifier in terms of prospective runtimes ratios, scoring a 16% improvement on `default` runtimes. Feature importance scores as assigned by `RF` models identify the same top-10 attributes in both setups, with the predicted version of `root_RelSignRDBDiff` ranking at positions 4 and 5, respectively.

5 Implementing predictions in CPLEX

For the actual implementation of our trained predictors into CPLEX, we select an `SVM` model trained in the setting with `SVR` regression performed on `Selected` features and `WTarLoss` used as custom

Table 8: Results for the binary setting using the `Selected` feature subset and predicted feature `root_RelSignRDBDiff` (21+1) from `SVR`, with sample weights w_{time} and custom cross-validation loss function `WTarLoss`. Reported values are averages across five experiments of the same type, on different seeds.

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

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

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

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

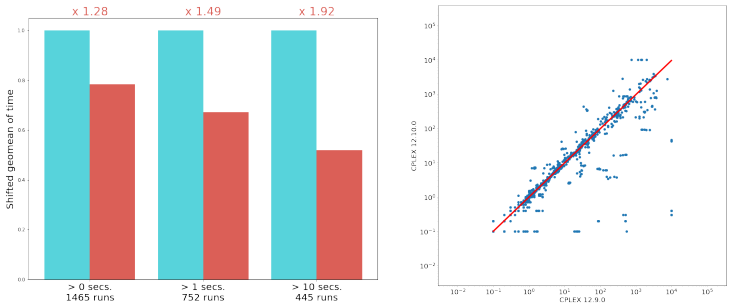
scoring function (cf. Table 8b). Overall, both SVM and RF models performed consistently well in our multiple experiments, but we ultimately opt for a SVM model over a RF one because of its easy-to-implement decision functions, whose coefficients and support vectors can be directly extracted with Scikit-learn.

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

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

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

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



(a) (b)

Figure 4: Comparison of MIQPs runtimes between CPLEX 12.9.0 and CPLEX 12.10.0 on the test set: (a) bar plot (CPLEX 12.10.0 in red), (b) scatter plot of running times in log scale.

select the final SVM model from a last round of training, and embed its decision function in the solver.

As a final experiment, we run both CPLEX versions 12.9.0 and 12.10.0, the latter incorporating the learned classifier, which is run by default, on the test set (on which the classifier was not trained) over five seeds. Note that the solving environment now presents some differences with respect to the setting in which we computed targets: for labeling, version 12.8.0 was used together with a time limit of 2 hours; now the time limit is raised to 10,000 seconds, and computations run on a more powerful cluster of identical machines with 16 core Intel X5650 processor at 2.67GHz, 24 GB RAM and using 12 threads. The classifier ultimately yields a 28% improvement of running time over the previous default strategy; the measure increases to 92% when considering runs[4] taking more than 10 seconds to solve with either version (Figure 4a). Figure 4b reports a scatter plot of MIQPs running times between versions 12.9.0 and 12.10.0. While using the classifier results in slower runs for some models, the degradation is generally limited and compensated by improvements of several order of magnitudes. In particular, only two runs present a degradation of more than one order of magnitude (the worst case being 13 times slower), while 58 (resp. 30) of them show an improvement of more than one (resp. two) order of magnitude.

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

6 Conclusions

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

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

and addressed it with ML techniques. The developed framework aims at embedding a predictive function in CPLEX: with this goal in mind, we contributed a methodological process for combining ML and MIP technology, and thoroughly revised our initial work [6]. We built a dataset of synthetic and real-world instances, proposing labeling schemes and carefully engineering features to describe MIQPs and the decision of whether to linearize them. Learning experiments as well as evaluation metrics were designed to integrate the optimization knowledge in the learning pipeline. In particular, we experimented with runtime weights, a custom scoring function and with the regression of an attribute about the root node bounds. Finally, we carefully considered how to include a predictor in the solver ecosystem. As a result, a SVM classifier deciding on MIQP linearization is implemented in CPLEX 12.10.0, establishing to the best of our knowledge the first example of a learning-based tool deployed in a commercial optimization solver.

References

[1] Warren P. Adams, Richard J. Forrester, and Fred W. Glover. Comparisons and enhancement strategies for linearizing mixed 0-1 quadratic programs. *Discrete Optimization*, 1(2):99 – 120, 2004.

[2] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: a methodological tour d'horizon. *arXiv preprint arXiv:1811.06128*, 2018.

[3] Alain Billionnet, Sourour Elloumi, and Amélie Lambert. Extending the qcr method to general mixed-integer programs. *Mathematical programming*, 131(1-2):381–401, 2012.

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

[5] Christian Bliek, Pierre Bonami, and Andrea Lodi. Solving mixed-integer quadratic programming problems with IBM-CPLEX: a progress report. In *Proceedings of the Twenty-Sixth RAMP Symposium*, pages 171–180, 2014.

[6] Pierre Bonami, Andrea Lodi, and Giulia Zarpellon. Learning a classification of mixed-integer quadratic programming problems. In Willem-Jan van Hoeve, editor, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 595–604, Cham, 2018. Springer International Publishing.

[7] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[8] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.

[9] CPLEX, 2020. `https://www.ibm.com/analytics/cplex-optimizer`.

[10] Joseph Czyzyk, Michael P. Mesnier, and Jorge J. Moré. The NEOS server. *IEEE Journal on Computational Science and Engineering*, 5(3):68——75, 1998.

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

[12] Robert Fourer. Quadratic optimization mysteries, part 1: Two versions, 2015. `http://bob4er.blogspot.ca/2015/03/quadratic-optimization-mysteries-part-1.html`.

[13] Fabio Furini, Emiliano Traversi, Pietro Belotti, Antonio Frangioni, Ambros Gleixner, Nick Gould, Leo Liberti, Andrea Lodi, Ruth Misener, Hans Mittelmann, Nikolaos Sahinidis, Stefan Vigerske, and Angelika Wiegele. QPLIB: A library of quadratic programming instances. *Mathematical Programming Computation*, 2018.

[14] William Gropp and Jorge J. Moré. Optimization environments and the NEOS server. In Martin D. Buhman and Arieh Iserles, editors, *Approximation Theory and Optimization*, pages 167–182. Cambridge University Press, 1997.

[15] Frank Hutter, Lin Xu, Holger H. Hoos, and Kevin Leyton-Brown. Algorithm runtime prediction: Methods & evaluation. *Artificial Intelligence*, 206:79–111, 2014.

[16] Elias Khalil, Pierre Le Bodic, Le Song, George Nemhauser, and Bistra Dilkina. Learning to branch in mixed integer programming. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence*, 2016.

[17] Andrea Lodi. Mixed integer programming computation. In M. Jünger, T.M. Liebling, D. Naddef, G.L. Nemhauser, W.R. Pulleyblank, G. Reinelt, G. Rinaldi, and L.A. Wolsey, editors, *50 Years of Integer Programming 1958-2008*, pages 619–645. Springer Berlin Heidelberg, 2009.

[18] Andrea Lodi and Andrea Tramontani. Performance variability in mixed-integer programming. In *Theory Driven by Influential Applications*, pages 1–12. INFORMS, 2013.

[19] Gilles Louppe. *Understanding Random Forests: From Theory to Practice*. PhD thesis, Université de Liège, Liège, Belgique, 2014.

[20] Garth P. McCormick. Computability of global solutions to factorable nonconvex programs: Part I — convex underestimating problems. *Mathematical Programming*, 10(1):147–175, 1976.

[21] MINOA. *Open-source benchmark library*, 2019 (accessed December, 2019).

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

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

Appendix

We include additional tables on MIQP features and their importance.

Table 9: Description of features in the `Initial` set (60). Features marked with * are part of the `Selected` set.

	Name	Description
	Static features	
	`RSizes`	Ratio of sizes m/n
*	`RBin`	Ratio of binary variables, over n
*	`RContInt`	Ratio of continuous and integer variables, over n
	`RNnzDiagBin`	Ratio of non-zero (nnz) coefficients in Q diagonal for binary variables
*	`RNnzDiagContInt`	Ratio of nnz coefficients in Q diagonal for continuous and integer variables
	`DiagDensity`	Density of diagonal entries of Q
*	`OutDiagDensity`	Density of non-diagonal entries of Q
*	`QDensity`	Density of Q
*	`RBinBin`	Ratio of nnz products between binary variables in Q
*	`RContContIntInt`	Ratio of nnz products between continuous or integer variables in Q
	`RMixedBin`	Ratio of mixed-type products involving binaries
	`RMixedContInt`	Ratio of mixed-type products involving continuous and/or integers
*	`RNonLinTerms`	Ratio of nnz non-linearizable terms, over n^2
	`RNonLinTermsNnz`	Ratio of nnz non-linearizable terms, over nnz
*	`RelVarsLinInc`	Relative size increase of potential linearization, over n
	`RelConssLinInc`	Relative size increase of potential linearization, over m
*	`RLinSizes`	Sizes m/n ratio after potential linearization
*	`NormMaxDegBin`	Maximum connectivity degree in Q among binary variables, over $n-1$
*	`NormMaxDegContInt`	Maximum connectivity degree in Q among continuous and integer variables, over $n-1$

Table 9 – continued from previous page

	Name	Description
	AvgDiagDom	Averaged "diagonal dominance" on rows [6]
	RDiagCoeff	Ratio of biggest and smallest diagonal nnz coefficients of Q, in absolute value
	ROutDiagCoeff	Ratio of biggest nnz diagonal coefficient and smallest out diagonal one, in absolute value
	RNnzBinLin	Ratio nnz binary variables in linear term
*	RNnzContIntLin	Ratio nnz continuous and integers variables in linear term
	HasLinearTerm	Boolean, whether there is a linear term
	LinDensity	Density of the linear term
	RLinCoeff	Ratio of biggest on smallest linear coefficients, in absolute value
*	ConssDensity	Density of constraints matrix A
	RConssBin	Ratio of constraints involving binary variables, over m
	RConssCont	Ratio of constraints involving continuous variables, over m
*	RConssInt	Ratio of constraints involving integer variables, over m
	RConssCoeff	Ratio biggest on smallest nnz constraints coefficients, in absolute value
	RRhsCoeff	Ratio magnitudes smallest on biggest nnz rhs coefficients, in absolute value
	RQTrace	Trace of Q, over n
	QSpecNorm	Spectral norm of Q
*	RQRankEig	Rank of Q over n (i.e., ratio of nnz eigenvalues of Q)
*	HardEigenPerc	Portion of problematic (hard) eigenvalues in Q
	AvgSpecWidth	Width of Q spectrum, over n
	RPosEigen	Ratio of positive eigenvalues, over n
	RNegEigen	Ratio of negative eigenvalues, over n
	RZeroEigen	Ratio of zero eigenvalues, over n
	RAbsEigen	Ratio of min and max eigenvalues, in absolute value
	RNZeroEigenDiff	Ratio of difference between original and corrected eigenvalues
	HardEigenPercDiff	Ratio of difference between original and corrected hard eigenvalues
	Preprocessing features	
*	prep_RelVarsIncL	Relative variables increase after L preprocessing
	prep_RelVarsIncNL	Relative variables increase after NL preprocessing
*	prep_RelConssIncL	Relative constraints increase after L preprocessing
	prep_RelConssIncNL	Relative constraints increase after NL preprocessing
*	prep_RSizesL	Sizes m/n ratio after L preprocessing
	prep_RSizesNL	Sizes m/n ratio after NL preprocessing
*	prep_ConssDensityL	Density of constraints matrix after L preprocessing
	prep_ConssDensityNL	Density of constraints matrix after NL preprocessing
	prep_ConssDensityDiff	Difference of density of constraints after preprocessing, between L and NL
	prep_RelConssDensityL	Relative density of constraints after L preprocessing with respect to original one
	prep_RelConssDensityNL	Relative density of constraints after NL preprocessing with respect to original one
	Root node features	
	root_RtTimeDiff	Difference of total root times (comprising preprocessing), between L and NL
	root_RLPTimeDiff	Difference of LP root times, between L and NL
	root_SignRDBDiff	Sign of dual bounds at root (1 if L better, -1 if NL better)
	root_RelRDBDiff	Relative difference of bounds at root
	root_RelSignRDBDiff	Signed relative difference of L and NL bounds at root

Table 10: Top-10 features identified by RF importance scores, in the multi-class and binary setting with Initial and Selected features. The reported scores are averages across the five RF models trained in each configuration.

(a) MultiLabel - Initial

Rank	Score	Feature
1.	0.1826	root_RtTimeDiff
2.	0.0897	prep_ConssDensityL
3.	0.0862	prep_RelVarsIncL
4.	0.0779	root_RelRDBDiff
5.	0.0512	root_RelSignRDBDiff
6.	0.0407	prep_RSizesL
7.	0.0391	prep_RelConssDensityL
8.	0.039	root_RLPTimeDiff
9.	0.0311	QDensity
10.	0.0293	OutDiagDensity

(b) BinLabel - Initial

Rank	Score	Feature
1.	0.0762	QDensity
2.	0.07	RBinBin
3.	0.068	root_RtTimeDiff
4.	0.065	OutDiagDensity
5.	0.0545	root_RelSignRDBDiff
6.	0.0419	NormMaxDegBin
7.	0.0327	prep_ConssDensityL
8.	0.0314	prep_RelVarsIncL
9.	0.0257	prep_RelConssDensityL
10.	0.0252	root_RelRDBDiff

(c) MultiLabel - Selected

Rank	Score	Feature
1.	0.203	prep_RelVarsIncL
2.	0.1807	prep_ConssDensityL
3.	0.1113	prep_RSizesL
4.	0.0571	QDensity
5.	0.055	RelVarsLinInc
6.	0.0519	prep_RelConssIncL
7.	0.0519	OutDiagDensity
8.	0.0513	RBinBin
9.	0.0441	NormMaxDegBin
10.	0.0306	RQRankEig

(d) BinLabel - Selected

Rank	Score	Feature
1.	0.1335	QDensity
2.	0.1177	OutDiagDensity
3.	0.1026	RBinBin
4.	0.0774	prep_RelVarsIncL
5.	0.067	NormMaxDegBin
6.	0.0633	prep_ConssDensityL
7.	0.0582	RelVarsLinInc
8.	0.0559	prep_RSizesL
9.	0.054	RQRankEig
10.	0.0453	RNonLinTerms