Karlsruhe Institute of Technology

Test, Validation, Analysis

# Performance Prediction for Subroutines in Meta-Solver Strategies

Bachelor's Thesis
of

**Piotr Malkowski**

at the Department of Informatics
Institute of Information Security and Dependability
Test, Validation and Analysis (TVA)

| | |
|---|---|
| Reviewer: | Prof. Dr. Ina Schaefer |
| Advisors: | M.Sc. Domenik Eichhorn |

# Contents

# 1 Introduction

Optimization problems that are widely recognized, such as scheduling, route planning, and resource allocation, continue to hold significance in both research and industry. A primary challenge when solving these types of problem is their NP-hard nature, which means that there are no known classical algorithms which can find the optimal solution quickly, making them difficult to solve as the problem size increases. However, through decades of advances in classical hardware, smart heuristics, and other improvement techniques, they are often solvable [36]. Nevertheless, some problem classes remain computationally infeasible, keeping out of reach for classical methods [3].

These challenges could be addressed by the ever-developing field of quantum computing. Qubits utilize quantum mechanics properties, such as superpositions and entanglement, allowing them to scale up to exponentially better than binary bits in classical computers for specific tasks. These characteristics sparked the creation of novel quantum algorithms, which are believed to provide advantages through better scaling and/or approximations [18, 49], providing us with the possibility to solve problems which cannot be tackled by existing classical methods [17, 48, 60]. However, current hardware development and software limitations restrict the applications of quantum computing to very niche applications.

Despite limitations, to still leverage quantum computing's potential when solving hard problems, hybrid-quantum-classical algorithm frameworks have emerged [15, 29]. The ProvideQ toolbox [15] enables users to easily test multiple hybrid Meta-solving strategies, with the choice of strategy left to the user. Meta-Solving formulates a strategy that breaks down complex problems, such as *Quadratic Terms with Binary Decision Variables* (QUBO), into manageable subproblems that can be solved using appropriate tools. Determining whether a step requires a certain quantum or classical approach relies on guesswork or intuition, expert knowledge and trial-and-error, leading to inefficiencies. Furthermore, each use-case may have different priorities: for problems with binary results - where answers are strictly correct or incorrect - the solver's runtime is critical. In other cases, approximate solutions within a specific confidence interval, meaning of appropriate solution quality, are adequate. Often, both quality and runtime are equally important.

This motivates *Empirical Performance Models* (EPM) [27], which are supervised machine learning models that can address these challenges by selecting the optimal solver for a given problem instance or by predicting solvers performance, such as runtime or solution quality, in advance. This can potentially save valuable time and resources.

For classical algorithm run-time prediction we rely on the paper "Algorithm Runtime Prediction: Methods & Evaluation" by Hutter et al. [27], who introduced the term "Empirical Performance Models" (EPM) in 2014. Their meta-study analyzed runtime prediction for three NP-hard problems - SAT, TSP and MIP - and compared different ML models performance on multiple datasets. They also provided valuable insights on how problem instances should be represented for the ML model to achieve the best results. Furthermore, Hutter et al. outlined a significant problem when obtaining algorithm runtimes for NP-hard problems: due to possible infinite

solver calculations, they had to cap the calculation at one hour, resulting in *censored* runtimes for some instances, meaning only the lower bound of the solver's termination time is known. They discussed methods for handling these censored observations, most notably the *survival analysis* methodology, which explicitly teaches the model to consider the censored characteristic. Hutter et al.'s publication is still the most cited and largest meta-study to date, directly aligning with our goal of runtime prediction for meta-solving subroutines. We reviewed other publications on EMP for algorithm runtime prediction such as [4, 47], however they all build on the foundational publication by Hutter et al. and directly address specific use cases or subcategories of EPMs.

Regarding ProvideQ quantum-classical subroutines, we are researching the application of machine learning (ML)-based performance prediction methods for hybrid quantum-classical solvers. The popularity of the hybrid quantum-classical algorithm framework is growing; however, the field of quantum-classical algorithm performance prediction is not well researched. Available publications on ML-based performance prediction for hybrid solvers [40, 61] focus on *automatically* selecting the best quantum-classical solver for a given problem instance without clarifying the reasoning behind the decision, which gives the process a black-box character.

In this thesis, we tackled the challenge of applying performance prediction to Meta-Solver Strategies by tackling three major Research Questions (RQs):

In RQ1, we revisited, explained, reproduced, and verified the decade-old EPM pipeline by Hutter et al. for the classical combinatorial optimization MIP problem. In RQ2, we modernized this pipeline by porting it to a modern programming language, implementing the state-of-the-art machine learning model XGBoost, training it with a modern, representative MIP dataset, and adapting it to survival analysis to better handle censored data. Lastly, in RQ3, we contribute to ML-based performance prediction methods for hybrid quantum-classical algorithms. We do so by explaining the current state of the literature and describing a novel method which utilizes ML to guide users in selection between different algorithmic approaches - quantum-classical, hybrid quantum-classical, and purely classical algorithms, with run-time and solution quality metrics.

**Readers Guide**

**In Section 2**, we offer an overview of the abstract terms and notations used in this thesis, with the goal of enabling the readers with background in either Quantum Computing or Machine Learning to follow this thesis without the need for external references. Next, we introduce the four major contributions of this thesis.

**In Section 3**, we describe machine learning models used throughout this thesis. Other publications that handle Empirical Performance Models often only briefly describe the used ML models and dive directly into evaluation. Our goal in this section is to provide deep, intuitive understanding of the machine learning models by presenting fundamental concepts of tree-based models, such as Regression Trees, Random Forest, Gradient Boosting and XGBoost. Furthermore we describe intuitively the Survival Analysis methodology, which allows ML models to learn from the *censored* data. Lastly, we justify with meta-studies the selection of the XGBoost as the model we use for subsequent improvements to runtime prediction.

**In Section 4**, we distinguish our thesis from other publications addressing classical algorithm runtime prediction using empirical performance models (EPMs), by revisiting the decade-old study by Hutter et al. While many publications build on the Hutter et al. study, delving into specific implementations, few verify the results presented. Our goal in integrating EPMs into subroutines in the Meta-Solver strategies provided by ProvideQ is to explore a new, under-researched use case. This leads us to believe that beginning with a foundational publication is beneficial. First, we summarize the Hutter et al. paper. Then, we describe and analyze the datasets, dynamic and static explanatory variables, data pre-processing, and evaluation of the machine learning models proposed by the authors. Next, we explain the changes the authors made to the default implementation of the best-performing model in their study: Random Forests, which achieved predictive error of **×4.37** the actual recorded run-times. Lastly, we update the authors legacy code, which no longer works, and reproduce their results on MIP a decade later. Surprisingly, through re-compilation of the code, we improved the results on the most important metric, Root Mean Square Error (RMSE), by 3.1%.

**In Section 5**, we present improvements to the EPM pipeline by Hutter et al., which we introduced in the previous section. Since the publication of the paper by Hutter et al. in the year 2014, ML standards have improved, with the introduction of state-of-the-art machine learning models XGBoost in year 2016 [11] and optimization frameworks OPTUNA in year 2019 [2]. The solvers used in the Hutter et al. study have since improved dramatically, yielding performance that is orders of magnitude better than a decade ago. Many instances that were considered unsolvable a decade ago can now be solved in a matter of seconds [37]. Furthermore, the spectrum of real-world applications and the difficulty of MIPs have evolved, prompting the need for modern MIP libraries such as MIPLIB [21] for evaluations that reflect the current state of the industry. Building on previous contributions and modernization, we improve the foundational study of EPM runtime prediction for classical combinatorial optimization problems by Hutter et al. To this end, we transition from the MATLAB programming language used by Hutter et al. to the modern Python programming language, which is the standard in current ML-space. We updated the machine learning model to XGBoost, optimized it with the automated hyperparameter search framework OPTUNA, implemented the modern dataset MIPLIB, and improved handling of censored algorithm runtimes with the survival analysis objective provided by the XGBoost model. Our optimized XGBoost model achieved an improvement of 8.1% on RMSE. Furthermore, we report how the optimized XGBoost model performs on a representative MIPLIB dataset where the predictive error is, on average, **×8.6** greater than the actual recorded runtimes. Finally, we demonstrate that XGBoost with survival analysis increases the rate at which the model detects timeouts from 7.7 to 32.1%, while decreasing the rate of false timeout predictions from 90 to 78.6%. We then argue why this is a reasonable trade-off when dealing with NP-hard MIP instances.

**In Section 6**, we analyze publications that address empirical performance models for solver selection in hybrid quantum-classical frameworks. We also address the lack of ML-based performance prediction methods that allow users to make informed decisions instead of making decisions for them. We draw inspiration from the publication by Moussa et al. [40], which proposes ML for selecting algorithms for classical and quantum-classical solvers (Goemans-Williamson and QAOA) for Max-Cut instances,

as well as the publication by Volpe et al. [61] that implemented a ML model to select the optimal classical or quantum-classical algorithm (QA, SA, QAOA, VQE, or GAS) for a given QUBO instance as part of the MQT Quantum Auto-Optimizer [64]. Finally, we describe how runtime and solution-quality regression could be used by ProvideQ users in selection between different algorithmic approaches, filing the gap in the ML-based performance prediction methods.

**In Section 7**, we describe the related work used in Sections 3, 4, 5 and 6. We provide brief descriptions of the publications we analyzed or drew inspiration from, as well as an outline of how our work distinguishes itself from others.

**In Section 8**, we summarize the four contributions, provide practical guidance on using EPM-based runtime prediction in ProvideQ, and outline future work - this includes extending EPMs to other subroutines and adding solution-quality prediction.

# 2 Background

This section offers an overview of the abstract terms and notations used in this thesis. Detailed methodological explanations can be found in the corresponding dedicated sections. Our goal is to share top-level vocabulary so readers with backgrounds in either Quantum Computing or Machine Learning can follow along without needing external references. We introduce Meta-Solving, Integer Programming, and Machine Learning, along with all the necessary subtopics.

We reference all the key concepts below in the appropriate places in the thesis for quicker navigation, if needed.

## 2.1 Meta-Solving & ProvideQ

Relevant optimization problems, such as scheduling and route planning, are challenging to solve due to their NP-hard nature. Advancements in modern classical algorithms often make them solvable [36], but some problem classes remain computationally infeasible [3].

Quantum computing could address this task. Through qubits, quantum computing utilizes quantum mechanical properties that quantum algorithms then use to solve problems that existing classical methods cannot tackle [17, 48, 60].

Despite limitations, to still leverage quantum computing's potential when solving hard problems, hybrid-quantum-classical algorithm frameworks have emerged [15, 29]. The ProvideQ toolbox [15] is software that allows users to easily test multiple hybrid solving approaches and get the benefits of classical and quantum computing by introducing meta-solvers. Available classical optimization algorithms include the Traveling Salesperson Problem, Knapsack, Max-Cut, Boolean Satisfiability (SAT) and Quadratic Unconstrained Binary Optimization (QUBO). The quantum solvers in the toolbox are based on the QAOA or Grover's algorithm.

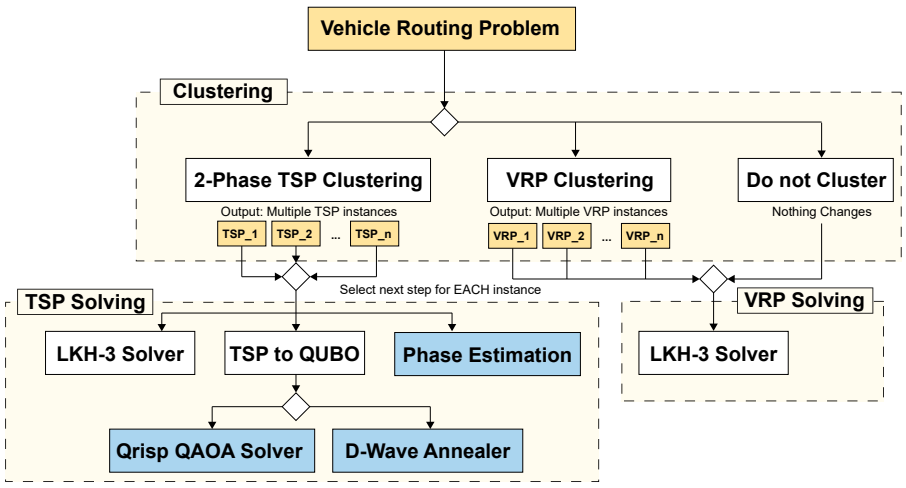



Figure 1: Running Example for a Vehicle Routing Meta-Solver Strategy [16].

In Figure 1, we observe meta-solving in action. We consider a specific subroutine: the Vehicle Routing Problem (VRP). First, the VRP is broken down into multiple smaller VRP problems via clustering. Then, each subproblem is solved using

subroutines such as the VRP or the traveling salesperson problem (TSP). Here, we focus on the relevant quantum-classical part: transforming TSP into quadratic unconstrained binary optimization (QUBO).

The resulting QUBO serves as standard input for quantum solvers, such as the Qiskit QAOA Solver or the D-Wave annealer, as illustrated in Figure 1. However, this transformation is quite resource-intensive due to the lack of constraints that must be encoded into the matrix.

According to Eichhorn et al., the QUBO subroutine is motivated by the fact that noisy intermediate-scale quantum (NISQ) hardware supporting algorithms such as QAOA could be available within the next few years, providing a competitive edge in solving optimization problems.

Next, we will introduce (Mixed-) Linear Programming, a subroutine available in ProvideQ that plays a central role in this thesis.

## 2.2 Integer, Mixed and Linear Programming

Linear Programming (LP) [8] is a fundamental combinatorial optimization approach in computer science. Formally, an LP consists of the following building blocks:

1. Variables, which can take real (continuous) values.
2. Linear constraints, expressed as (in-)equalities over sums of variables scaled by coefficients.
3. A linear objective function to be maximized or minimized.

A basic example of a linear program in standard form could be expressed as:

$$\begin{aligned} \max \quad & 40\,x_A + 50\,x_B \\ \text{s.t.} \quad & 2\,x_A + 1\,x_B \leq 100, \\ & x_A,\, x_B \geq 0. \end{aligned}$$

This basic LP example can be described as: "How much of products A and B should be produced to maximize profit (or minimize cost)?" The variables, $x_A$ and $x_B$ represent the quantities of products A and B to produce. The constraint models resource usage: producing one unit of A consumes 2 hours, and B consumes 1 hour, with a total production time limited to 100 hours. The objective function $40x_A + 50x_B$ maximizes profit, assuming each unit of A yields 40€ and B 50€.

This basic example can be easily solved by hand. However, this structure yields a compact model whose solution, in more complex cases, can be efficiently found using modern LP solvers like Gurobi or CPLEX.

LPs belong to the complexity class $P$. Khachiyan [34] has found, that they can be solved in polynomial time with Ellipsoid method. However, this approach is rarely used because of the poor performance. In practice, Simplex algorithm [35, 50] is used, which has exponential worst-case complexity, but performs typically well on average.

One instance where complex cases arise are when decisions must be integer-valued, e.g., "how many whole trucks to send on a route". The model moves then from LP to Integer or Mixed Integer Programming (ILP and MIP) [42].

Although integer constraints may seem like a simple restriction (as integers are a subset of real numbers), they break the convexity of the problem. This change moves the problem from the $P - hard$ to $NP - hard$ class of combinatorial optimization problems, because it requires the solver to explore many discrete combinations of feasible integers. In the worst-case, this can require exponential time in the number of variables, although they can be still efficiently solved with the aforementioned Simplex and branch-and-bound algorithms.

Modern solvers employ advanced heuristics and optimization techniques that enable solving many large MIP instances in practice. Still, some instances remain notoriously difficult to solve to optimality, leading to high variance in runtime and solution quality.

Understanding the properties of MIP instances, such as characteristics of constraints, tightness of bounds or number of integer variables, is currently still under research. These properties are used as explanatory-variables in machine learning models, allowing us to understand the connection between instances and the solver behavior.

In the following section we introduce the general concept of machine learning combined with all the necessary topics, such as explanatory-variables.

## 2.3 Machine Learning

Machine learning (ML) is a methodology in which computers and machines learn themselves from provided data as opossed to relying on the hard-coded rules, such as in programming. The goal of ML models is to imitate and provide domain expertise in tasks too complex for humans or rule-based systems [28].

The UC Berkley [59] describes three main components of ML. Machine Learning is a (i) decision process that maps input data to a continuous prediction or class with an error or loss function (ii) that scores how good the model is against unknown examples and a iterative learning / optimization process (iii) that adjusts the tunable parameters to reduce error until maximal performance is reached.

ML falls into 3 main sub-categories: supervised, unsupervised and reinforcement learning - each with different use-cases. In this thesis we use ML for supervised regression: we learn the correlation between explanatory-variables for optimization problems and solver behavior, for example predicting runtime.

### Classification and Regression

Classification and regression are two main categories in supervised machine learning. They differ by the type of prediction they produce. Classification predicts a discrete outcome, such as determining whether a given MIP instance will be solved faster with CPLEX or Gurobi. Regression, on the other hand, predicts a continuous value, for instance, estimating the runtime of a solver in seconds. Both classification and regression require a labeled training set of instances containing explanatory variables. These explanatory-variables are characteristics of the problem instance, such as the number of variables, the number of constraints, or the overall problem structure.

These models are then evaluated based on their ability to generalize to new, unseen data.


(a) Regression (b) Classification

Figure 2: Visualization of the two supervised learning objectives. (a) regression, where the objective is to predict a continuous value and (b) classification, where the objective is to assign the samples to a discrete category, here blue and red [43].

The difference can be well visualized in the Figure 2 above. In regression, given an input $x$ and output ("labels") $y$, the task is to learn a continuous function $y = f(x)$.

Classification, on the other hand, given an input $x$ and output-classes $c$, the task it to learn the class-assignment $y = f(x)$.

**Explanatory-Variables**

Pedro Domingos, in his highly cited publication "A Few Useful Things to Know About Machine Learning" [14], states that the success of an ML project depends on the learned explanatory-variables.

Note that the two main terms used in the literature are "features" and "explanatory variables." From this point on, we will primarily use the latter.

For instance, if one finds explanatory variables with linear dependencies on the algorithm's runtime, even the simplest ML models will perform brilliantly. However, if these variables are of very high dimensionality, with no clear insight into their correlations and predictive performance, even the best models will perform no better than guesswork. For example, if we wish to predict someone's final undergraduate grades, we can expect that characteristics such as the time they spend learning each day, the number of courses they participate in, or their engagement in lectures to be better indicators of performance than their height measured in cm, eye color, or the color of their house.

Furthermore, it is not feasible to train a model on thousands of explanatory variables with the hope that the model finds the ones that are relevant and discards the rest. This problem, known as the "curse of dimensionality", can be illustrated with the following example: consider an input space of 100 explanatory variables per instance and a dataset with 1 trillion instances. The dataset in this case covers only an extremely small fraction of the input space, about $10^{-18}$ [14]. This process of explanatory-variable search can be automated by first using many explanatory variables and then performing dimensionality reduction with unsupervised ML algorithms like PCA [30]. However, many explanatory variables that look useless in isolation can be highly descriptive in combination, adding further complexity to this topic. Andrew Ng, one of the most cited professors in ML, has stated in one of his lectures that "coming up with features is difficult, time-consuming, requires expert

knowledge. Applied machine learning is basically feature engineering." [58] From this quote, we can deduce that domain insight is a major part of ML and outweighs data-science tricks.

**Dataset Splits**

**Train-Test**

An ML model should not be trained on the entire available dataset, because otherwise we would not have a reference for how the model performs on unseen data. For example, during training, it could report incredible (or terrible) accuracy (performance) measured by the loss function, which is a mathematical formula which quantifies the difference between models prediction and the target-values, but then subsequently perform no better than guesswork when deployed on previously unseen data.

The need to test on unseen data is connected to the concepts of bias and variance, which help to describe why a model has difficulties to generalize. Bias refers to errors made by overly simple models; high bias tends to underfit the training data, meaning that the model performs poorly on the training and test sets.

Variance, refers to a model's sensitivity to changes in the training set. A model with high variance tends to achieve great results on the training data but low scores on the test data, meaning it fails to generalize and thus overfits.

This can be described by the expected behavior of the prediction $\hat{f}_i$ with:

$$\mathbb{E}\left[\hat{f}_i\right] = \mu \quad \text{and} \quad \text{Var}\left[\hat{f}_i\right] = \sigma^2$$

In the formula above, $\mathbb{E}\left[\hat{f}_i\right] = \mu$ is the average prediction the model makes at input $x_i$ over many runs, and $\text{Var}\left[\hat{f}_i\right] = \sigma^2$ measures how sensitive the model's prediction for $x_i$ is to changes in the training data.

The default approach is to split the data-set into the training and test data (typically 80:20% split) [24]. This split can be done randomly to ensure that both sets are representative of the original dataset. However, in some cases, certain sample values in the dataset are underrepresented. For example, if these values represent only 1% of the total values, they would be underrepresented. In these cases, a stratified split is preferred to ensure the same proportions of data in both the training and test sets.

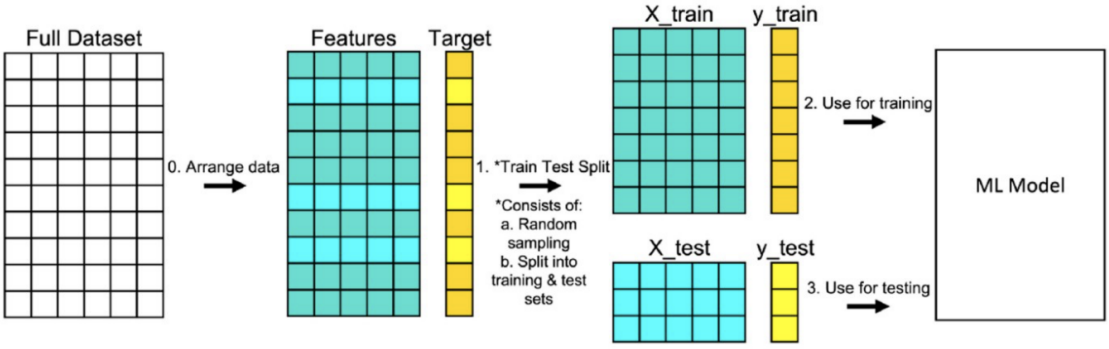



Figure 3: Visualization of train-test split approach: ordering the dataset into explanatory-variables and target, randomly splitting into training and test sets and using them for model training and evaluation [43].

As visualized in Figure 3, we train the model on the training dataset and then evaluate it on previously unseen data. But the question remains: we now have a reference for the actual performance - but what should we do if it is not as good as we hoped?

During training, the model strives to set weights such that the error, measured by the loss function, is as low as possible. However, we utilize regularization to encourage the model, or rather the loss function, to prefer less complex (i.e., lower-capacity) functions by penalizing overly large and thus overly specific parameters, in order to achieve optimal capacity as pictured below in Figure 4.

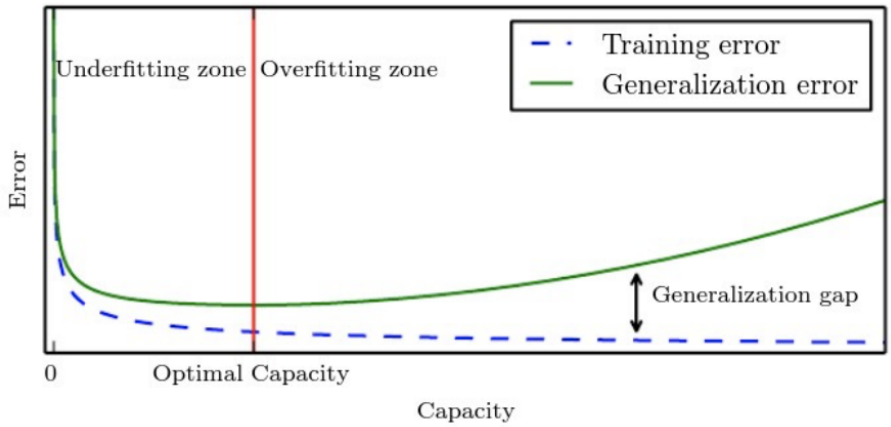



Figure 4: Trade-off between bias and variance: training error sinks with model capacity. However, generalization error forms a U-shape, which illustrates underfitting, optimal capacity and overfitting [43].

A further method by which we can improve the results is by adapting the architecture itself. For instance, how many layers and neurons per layer a neural network has, or how many trees, leaves, or what depth tree-based ML architectures grow to. All these parameters, which are not adapted during training but are set in stone at the very beginning, are called hyperparameters.

They, unfortunately, pose further difficulties. Consider an example: we evaluate the hyperparameters at the same time as the model parameters during training, given this regularized mean squared error loss function:

$$L(w) = MSE_{train} + \lambda w^T w \; with \; MSE_{train} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Here, $MSE_{train}$ is the typical loss used in regression, with $y_i$ and $\hat{y}_i = x_i^T w$ true target values and model predictions respectively, $w$ the model weights and $n$ number of training samples. The term $\lambda w^T w$ is the L2 Regularization Term, which penalizes large weights, encouraging the model to keep the weights smaller and thus prevent overfitting and improve generalization on the unseen data.

If we were to optimize the hyperparameters, here $\lambda$, on the training-data, this would lead to larger, more specific weights and thus lower training error, but at the cost of increased overfitting and reduced generalization.

The solution is to further split the dataset specifically for validating the hyperparameters (compare Figure 5). Hence, the previous train:test 80:20% split becomes train:validate:test of, for instance, 60:20:20% proportions.

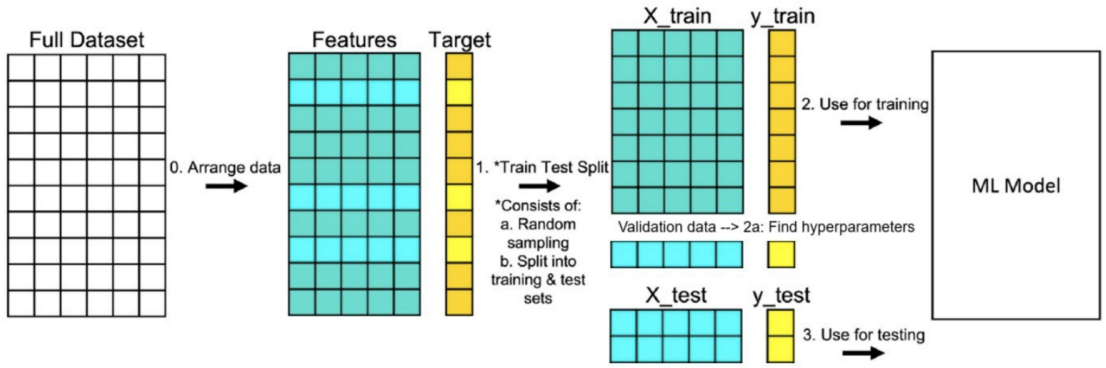



Figure 5: Train-test-validation approach: ordering the dataset into explanatory-variables and target, randomly splitting into training, validation and test sets and using them for model training, hyperparameter tuning and evaluation [43].

### Cross-Validation

The approach mentioned above is the industry standard; it is not, however, a silver bullet. Reserving almost half (40%) of the data for model training evaluation and hyperparameter optimization works well, assuming we work with large datasets. This is where cross-validation has established itself as a further industry standard [38].

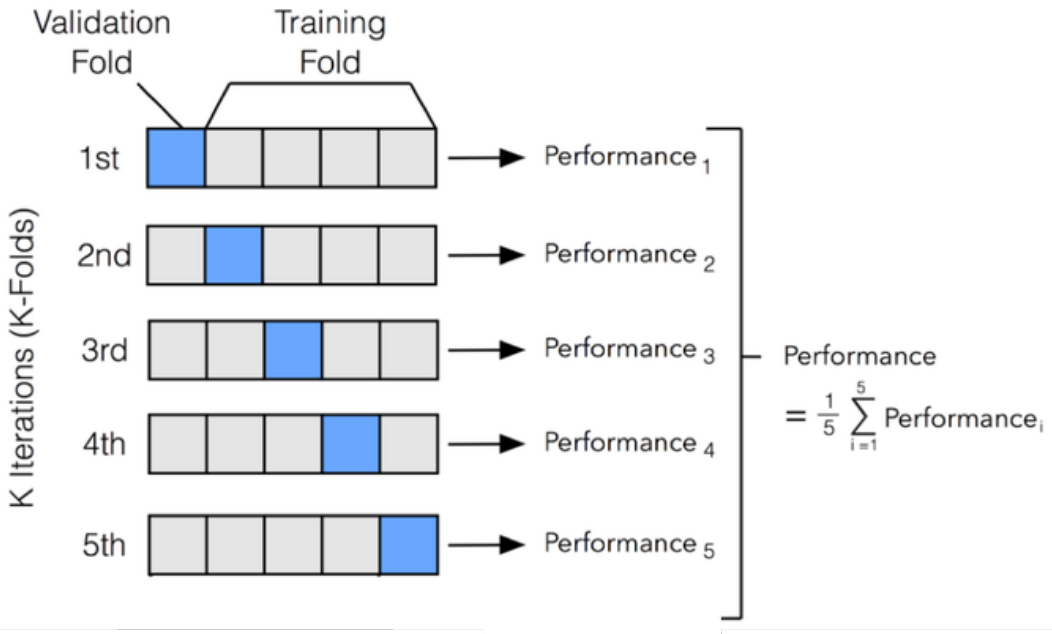



Figure 6: Visualization of 5-fold cross-validation. The dataset it split into five folds, where each fold is used once as validation data and the remaining folds create a training set. The final predictive performance is the average over all folds [43].

We optimize the hyperparameters by minimizing the validation error while utilizing different parts of the dataset during training. In Figure 6 above, we use a fifth of the data in each round, making it a ”5-fold cross-validation.” Consequently, we calculate performance as the mean across the k-iterations to obtain reliable results.

# 3 Classifying Machine Learning Models

This section provides a deeper understanding of the machine learning models that will be used in subsequent sections. Having intuitive and detailed knowledge of these models is necessary to understand the methodology and reasoning behind the subsequent sections experiments.

A significant part of the analysis in this thesis involves improving the run-time prediction methodologies presented in the literature. To this end, in the conclusion, we will use our detailed knowledge to compare the proposed machine learning models to determine which could yield the greatest improvement in our task of solver runtime prediction.

We have organized this section as follows: First, we provide a general overview of the model. Next, we illustrate use cases with a concrete example. Lastly, we offer an intuitive explanation of the mathematical details important for correct implementation.

We start with the fundamental concept of tree-based models and Regression Trees. Then, we describe how this concept progresses to ensemble methods, such as Random Forest, which reduce variance by averaging across multiple semi-independent regression trees. Next, we transition to gradient boosted trees, which improve performance by sequentially building trees that learn from the errors of previous trees. Lastly, we introduce XGBoost, the state-of-the-art architecture used for tabular data such as the explanatory variables of MIP instances and the run times as target values.

Furthermore, we delve into an important aspect of our use case: handling censored data. We introduce survival analysis, which enables the models to handle incomplete or terminated target values. This is relevant for many instances in our experiment because we used a solver time limit for hard instances.

## 3.1 Regression Trees

Regression Trees [9] are a supervised ML method that can be used for both regression and classification. Like Support Vector Machines (SVM) or Logistic Regression, they are a method that can automatically learn non-linear relationships, in contrast to linear regression, which can only be applied in simple domains (compare Figure 7).

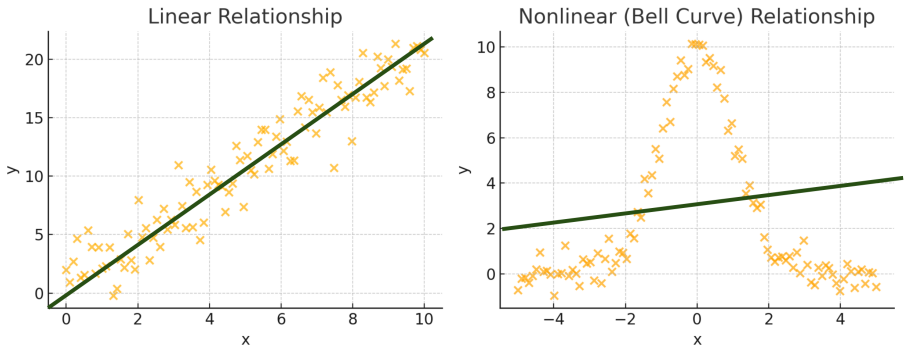
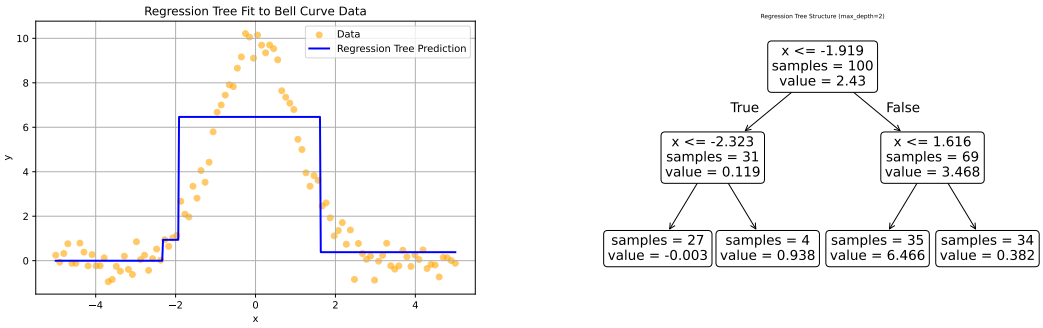



Figure 7: Visualization and comparison of a linear regression model on a dataset with linear relationship (left) and a nonlinear Bell Curve relationship (right). This comparison shows the limitations of linear regression when the data has nonlinear properties.

Regression Trees are a type of Decision Tree, where each leaf is assigned a numerical value, and each node is an if-statement based on explanatory variables that determines whether the tree is parsed to the left (for true) or to the right (for false).




(a) Fit (b) Structure

Figure 8: Visualization of a regression tree with depth 2 applied to a generated example nonlinear Bell Curve dataset. (a) represents the fitting-steps and predictions compared to observed data and (b) shows a corresponding tree structure with detailed decision splits and average target values in leaves.

In Figure 8 above, we can see how a regression tree with a depth of 2 would approach the nonlinear Bell Curve dataset. The predicted value in each leaf is simply the average of all observations that fall into it. The example above contains only one explanatory variable (x) to predict the value of the target variable (y). The strength of regression trees lies in their ability to handle high-dimensional data where learning the context by heuristics or domain experts would be impossible.

**Example**

Here we dive into how regression trees are built, following intuitive explanations provided by PhD Joshua Starmer, a former assistant professor at University of North Carolina and current CEO of StatQuest, which offers educational materials that teach data science, machine learning and statistics [51, 53].

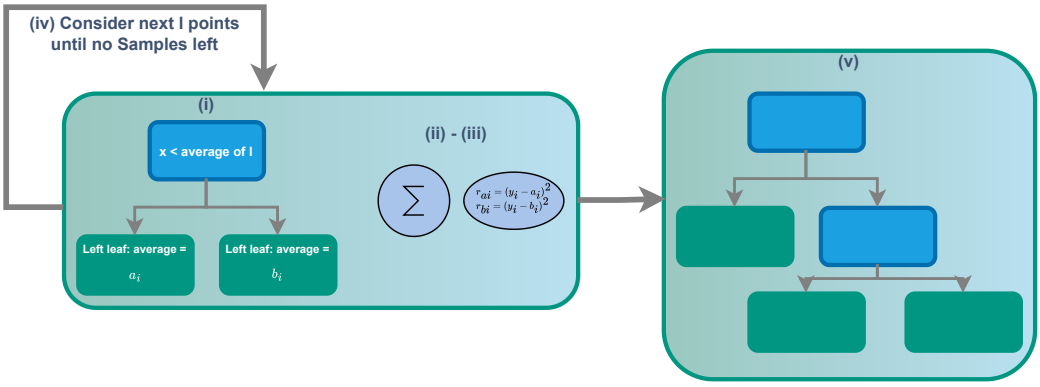



Figure 9: Regression Tree structure: create candidate stump splits at l-point averages of x, set leaf prediction to leaf means, score each split by sum of squared residuals and pick the treshhold achieved by the best split.

Once again, consider a dataset that contains only one explanatory variable $x$ and one target variable $y$. The initial step is to build a stump (i) (a tree containing only one root node and two leaves). For that, we take $l$ data points (samples) with the lowest values of the explanatory variable $x$ and build a tree where all points (initially only one) that are less than or equal to this averaged value of $x$ are sorted into the left leaf, and the rest into the right.

Afterwards (ii), for each leaf, we calculate the average of the $y$ values of the samples contained in the leaf and set it as a new prediction. Next (iii), we calculate the Sum of Squared Residuals (SSR) by adding up and squaring all the residuals $r$, which are the differences between the predicted and true values of $y$ for each sample. The result of this summation is an indicator of how well this threshold performs.

Subsequently (iv), we consider the next $l$ data points and repeat steps (i–iii) until there are no samples left to consider. Given all the loss function values for different thresholds of $x$, we choose the threshold with the lowest value.

Following that (v), we can grow the stump further by looking at nodes in the tree that contain multiple samples and splitting them using the methodology explained in steps (i–iv).

However, this repetition could likely result in overfitting on the training data, as we would grow a deep tree where each final leaf contains only one sample and would likely generalize poorly. To counter this, one method is to stop splitting a node unless it contains more than a user-specified number of samples (a hyperparameter). A further method is to implement cost-complexity pruning.

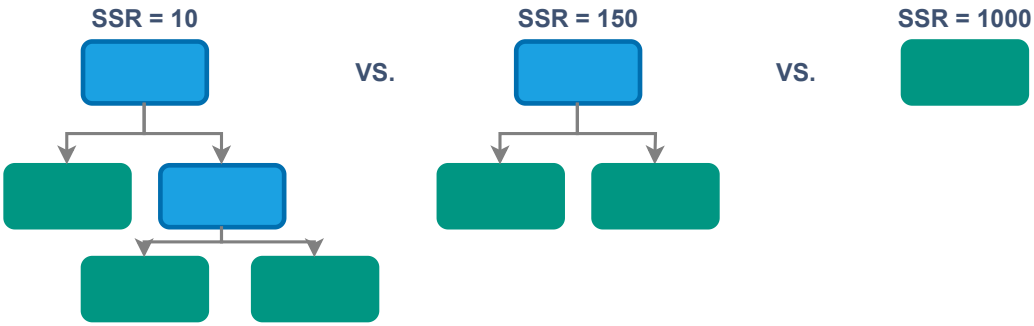



Figure 10: Cost-complexity pruning: compare candidate trees by total SSR (sum over nodes) and select the tree that minimizes $SSR + \alpha \times |leaves|$, with consideration that deeper Trees lower SSR but pay a complexity penalty $\alpha$.

For cost-complexity pruning, we merge the leaves bottom-up and calculate the SSR for the entire tree by adding up the individual SSRs for each node. Consequently, we compare the tree scores, which are the tree SSRs plus a complexity penalty for the number of leaves, and pick the one with the best (lowest) score.

The steps above explained how a tree is built for a single explanatory variable $x$. Usually, however, we train Regression Trees on datasets with higher dimensionality. For that, we apply steps (i–iv) to each of the explanatory variables and, at each step, choose the one that yields the best result.

**Mathematical Background**

Given a training sample $D = \{(x_i, y_i)\}_{i=1}^{n}$ with $x_i \in \mathbb{R}^p$ and $y_i \in \mathbb{R}$, a regression tree seeks a function

$$\hat{f}(x) = \sum_{m=1}^{M} \hat{c}_m \mathbb{I}(x \in R_m), \quad \text{where } \hat{c}_m = \frac{1}{|R_m|} \sum_{x_i \in R_m} y_i.$$

A regression tree can be imagined as splitting the visualized data space into rectangular regions $R_m$, which correspond to the leaves in the tree. Inside each region, every sample that falls into it receives the same prediction $\hat{c}_m$. This prediction is the average of the observed target values of all samples within $R_m$. We minimize the sum of squared differences (MSE) to get the value that is, on average, closest. Once the value $\hat{c}_m$ has been calculated for all regions in the tree (with $m \in M$, where $M$ is the number of leaves), the function $\hat{f}(x)$ is used to express all these regions.

The training objective, typically squared-error loss for regression, can be expressed as:

$$L(T) = \sum_{m=1}^{M} \sum_{x_i \in R_m} (y_i - \hat{c}_m)^2$$

where $T$ is the tree. Since minimizing this loss function over all possible partitions for each explanatory variable would result in exponential slowdown, trees are instead grown top-down, where the loss function is minimized greedily at each split.

This greedy split criterion can be expressed as follows: a node $t$ in the tree contains all samples from the subset $S_t$ of the data. For every feature $j$ and threshold $s$, we define the left and right child nodes as:

$$S_L(j, s) = \{(x, y) \in S_t : x_j < s\}, S_R(j, s) = \{(x, y) \in S_t : x_j >= s\}$$

The algorithm evaluates all possible pairs $(j, s)$ and selects the one that minimizes the squared error. Once such a node is found, the process is repeated recursively until a stopping criterion is met, for instance, reaching the maximum depth, failing to meet a minimum node size, or achieving zero residual error.

As mentioned above, a fully grown tree tends to overfit. To counter this, we use Cost Complexity Pruning, which prunes the tree bottom-up using the following function:

$$C_\alpha(T) = L(T) + \alpha|T|$$

where $|T|$ is the number of leaves, and the hyperparameter $\alpha$ controls the complexity penalty. Increasing $\alpha$ leads to smaller trees. The value of $\alpha$ can be determined through hyperparameter search with cross-validation to balance bias and variance, meaning to avoid both underfitting and overfitting.

Regression trees provide an intuitive approach to modeling nonlinear relationships. However, they have one major drawback: they are unstable, and small changes in the training data can result in different trees, leading to high variability in predictive performance. To address this issue, ensemble methods were developed to reduce variance. Below, we present the Random Forest methodology, which builds on regression trees.

## 3.2 Random Forest

Regression Trees are known for being difficult to balance between overfitting and good performance. This is why Random Forests [25], ensembles of regression trees, have emerged.

The process can be defined with the following steps: (1) First, we create a bootstrapped dataset by randomly sampling from the original set, with the quirk that the same sample can appear multiple times in the bootstrapped dataset.

(2) Then, we create a regression tree (compare subsection 3.1), but at each split point, we consider only a random subset of the explanatory variables. These two aforementioned steps are where the "random" in Random Forests comes from.

We repeat steps (i) and (ii) until the desired number of trees has been reached, which results in the structure depicted below in Figure 11:

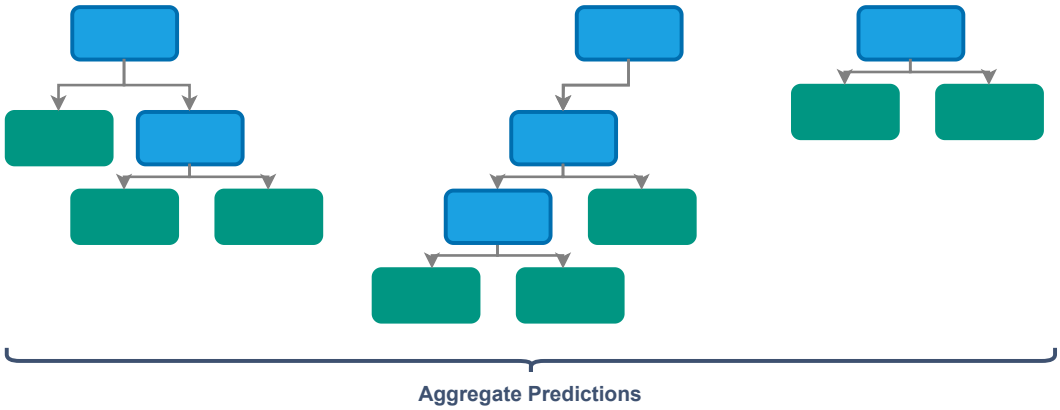



Figure 11: Random Forest structure: multiple Regression Trees are trained on bootstraped samples with random explanatory-variable subset. Final predictions is the average across trees (bagging) [51, 53].

Predictions are made similarly to those in regression trees. However, instead of relying on the output of a single tree, we aggregate the predictions from all trees by calculating their average. This process of bootstrapping and aggregating results is called bagging.

### Mathematical Details

We denote the prediction of a single tree for input $x_i$ as $\hat{f}_i$. If we train $m$ such trees on bootstrapped data and then aggregate the results by averaging the predictions, we get the following ensemble prediction:

$$\hat{f}_{ens} = \frac{1}{m} \sum_{i=1}^{m} \hat{f}_i$$

Assuming that each tree has the same expectation and variance, we have:

$$\mathbb{E}[\hat{f}_i] = \mu \quad and \quad Var[\hat{f}_i] = \sigma^2$$

Using the linearity of expectation, we obtain:

$$\mathbb{E}[\hat{f}_{ens}] = \mathbb{E}[\frac{1}{m}\sum_{i=1}^{m}\hat{f}_i] = \mu$$

This means that bagging leads to the same bias for both Regression Trees and Random Forests. However, if we examine the variance using the laws of scaled variables, we observe:

$$Var[\hat{f}_{ens}] = \frac{1}{m}\sigma^2$$

for independent models.

$$Var[\hat{f}_{ens}] = \frac{1}{m}(1-p)\sigma^2 + p\sigma^2$$

for correlated trees with correlation coefficient p. From the second formula, we see that as the correlation $p$ increases, the variance reduction becomes smaller, approaching the default $\sigma^2$. Random Forests reduce $p$ by introducing two sources of randomness:

- bootstraping of random samples
- random feature subset considered at each split

In conclusion, Random Forests address the primary weakness of Regression Trees: their tendency to overfit. However, this comes at the cost of increased model size and training time. However, they do not directly address bias because each tree is trained independently on bootstrapped data. Next, we will introduce a different methodology: the boosting method. This method builds trees sequentially, with each tree learning from and improving upon the errors of the previous trees. Thus, it iteratively optimizes the chosen loss function.

## 3.3 Regression Gradient Boosting

To better utilize the forest of trees presented in the previous chapter, Gradient Boosted Trees emerged [20]. They utilize a complex sequential building process that learns from the errors trees in previous iterations made.

Initially, a single leaf is constructed, which represents the assumed prediction, also called the target-value, for all samples in the training dataset. This target variable is the mean across all target values. Then, a tree is built without size restrictions. In the next iteration, another tree is built in the same fashion, but with consideration of the errors made by the previous tree - this sequential process is called boosting.

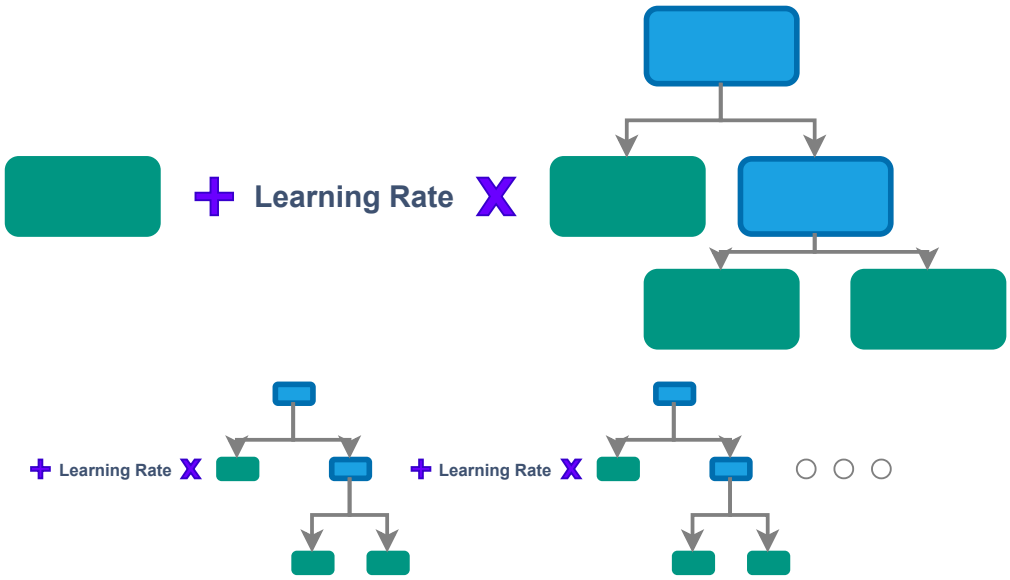



Figure 12: Gradient boosting with Regression Trees: start with a constant prediction, followed by iteratively fitting trees to residuals and lastly adding each trees scaled contribution (with the learning rate) to the model [51].

We see in Figure 12, that this procedure is repeated until a user-specified number of trees has been built or until further trees fail to improve the fit.

Below, we present an example of how the trees are built [51, 52].

**Example**

After building the first tree, we subtract the target variable from the assumed prediction for each sample to obtain the residuals (i).

Next, we use the explanatory variables to build a tree (compare subsection 3.1) that predicts these residuals (ii). We observe that a few data points, based on their explanatory variables, land in the same leaf when traversed through the tree. For instance, the first and fifth residuals may fall into the same leaf. We then replace the multiple residuals in that leaf with their average.

The last step (iii) is to take the predictions from the previous step (i), scale them with the learning rate (a value between 0 and 1), and add them to the new predictions from step (ii). The scaling is necessary, as it prevents the model from overfitting the training data, thus improving its ability to generalize (compare Figure 13).

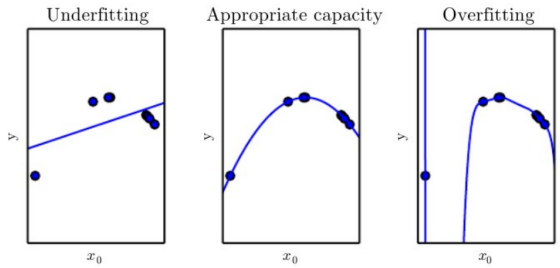



Figure 13: Visualization of observed data-points and fitted predicted values. Underfitting occurs when the model is too simple to fit the datastructure; Appropriate capacity represents balance between bias and variance; Overfitting happens when the model is to complex and fits the training data perfectly, but does not generalize to unseen data [43].

Now, we evaluate how well the tree performs by building new residuals for each sample using the combined predictions from all trees, and repeat the entire cycle. Each iteration is evaluated by comparing the residuals from the previous iteration. The premise is that by taking small steps in the right direction each time (as evaluated by comparing residuals), we converge to a local minimum, as described by Friedman et al. [20].

When the model is deployed, and given a new, previously unseen input $x$ (or rather its explanatory variables), we traverse each tree by descending through the if-statements until reaching the leaf nodes. Then, just like in the training phase, we sum the predictions from each tree, scaled by the learning rate, yielding a regressed value for the unseen input (ensemble prediction).

### Mathematical Background

We denote the dataset as $\{(x_i, y_i)\}_{i=1}^{n}$, where $x_i$ denotes the i'th sample's explanatory variables and $y_i$ the i'th target variable. Furthermore, we consider a differentiable loss function $L(y_i, F(x))$. Usually, for regression, we use $L(y_i, F(x)) = \frac{1}{2}(y_i - \hat{y}_i)^2$, where $y_i$ is the observed (available in the dataset) target variable and $\hat{y}_i$ the predicted target variable by the model, obtained by $F(x_i) = \hat{y}_i$. The intuition for the loss function can be understood by looking at Figure 14.

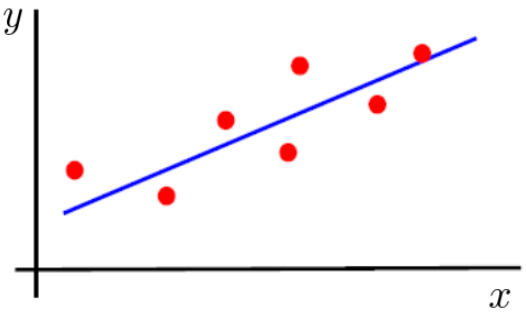



Figure 14: Visualization of simple regression. Red points are the observed data tuples $(x_i, y_i)$. The blue line represents the fitted regression function $\hat{y}_i = F(x_i)$. The vertical distance between the blue line and red points corresponds to residuals $r_i$, which are minimized by the squared loss function [43].

Consider the red data points, which denote the observed $y_i$, and the blue line, which shows our regressed (predicted) values $\hat{y}_i$. The difference for each sample is the residual $r_i$. The term is squared for the following reasons:

- non-linearity: to penalize large errors more.
- differentiable property: without the exponent, the function's derivative would be undefined in the $y_i = \hat{y}_i$ point.

The factor $\frac{1}{2}$ cancels the factor of 2 after taking the derivative with respect to the predicted value: $\frac{d}{d\hat{y}} \frac{1}{2}(y_i - \hat{y}_i)^2 = -(y_i - \hat{y}_i)$. This leaves us with the negative residual, which is trivial to calculate.

The first iteration of step (i) from Figure Y (leaf) can be denoted as $F_0(x) = \arg\min_{\hat{y}} \sum_{i=1}^{n} L(y_i, \hat{y})$. This step denotes summing the loss function for each observed value and then searching for such predicted values that minimize the overall residual. The $\arg\min_{\hat{y}}$ step is achieved by deriving the sum with respect to the predicted values $\hat{y}$, resulting in the negative residual $r$ as above.

Steps (ii) and (iii) involve the following algorithm:

for $m = 1$ to $M$ (with M being a number of trees to compute):

(A) Compute $r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)}$ for $i = 1, \ldots, n$

(B) Fit a regression tree to the $r_{im}$ values and create terminal regions $R_{jm}$, for $j = 1, \ldots, J_m$

(C) For $j = 1, \ldots, J_m$ compute

$$\gamma_{jm} = \arg\min_{\gamma} \sum_{x_i \in R_{jm}} L(y_i, F_{m-1}(x_i) + \gamma)$$

(D) Update

$$F_m(x) = F_{m-1}(x) + \nu \sum_{j=1}^{J_m} \gamma_{jm} \mathbb{I}(x \in R_{jm})$$

Step (A) calculates the positive residuals $r_{im}$, with $i$ denoting the sample and $m$ the current tree. The next step (B) involves calculating the regression tree (compare subsection 3.1), with the difference that the predicted values contained in the leaves (regions $R_{jm}$, with $j$ being the leaf) are not built to predict the target variable but the residuals calculated in step (A).

Step (C) addresses the issue of multiple numerical values contained in one leaf. This calculation is very similar to how $F_0(x)$ was computed, with the difference that the previous prediction $F_{m-1}$ is now taken into account to find the predicted values that minimize the residual for the specific leaf.

Finally, step (D) calculates the new prediction for each sample by adding the prediction from the previous tree to the predictions given by all leaves in the new tree

that contain the sample $x$. The hyperparameter $\nu \in (0, 1]$ denotes the learning rate, meaning the extent to which this tree contributes to the overall prediction in the ensemble.

In conclusion, gradient boosted trees are a powerful machine learning model. Next, we introduce Extremely Gradient Boosted Trees, which are based on the Gradient Boosted Trees method. However, they introduce a more regularized objective function, more complex tree-splitting criteria, and optimizations to further improve predictive performance.

## 3.4 XGBoost

The further well-established and highly competitive tree ensemble boosting method is eXtreme Gradient Boosting (XGBoost) [11]. It builds upon the concepts introduced in the Regression Trees (subsection 3.1) and Gradient Boosting (subsection 3.3) subsections but introduces a novel method for constructing regression trees.

#### Example

Just like Gradient Boosting, XGBoost is fitted on the residuals and not on the target variables. At first, each tree starts as a single leaf containing the residuals for all samples. However, unlike the squared residual difference used in subsection 3.3, XGBoost aggregates all residual values using a similarity score and a regularization parameter (i).

At this step, we aim to expand the tree further. The main question is whether the model can improve the similarity score by clustering the residuals into different groups. We split the groups by considering a splitting point defined by the average of a certain explanatory variable - this splitting point defines a cut-off barrier, thereby creating two new leaves (ii) [51, 52].

The next step is to calculate and evaluate the similarity scores across all three nodes in the tree. This is achieved using the gain score, where higher values indicate better information gain across the entire tree (iii). This process is repeated for different thresholds defined in step (ii). Each iteration is compared using the gain metric - until the threshold can no longer be improved. By this point, the process has found the best threshold for a certain explanatory variable in the dataset, creating the first branch in the tree (iv).

Steps (i–iv) are repeated until there is only one residual per node (no further split is possible) or a predefined maximum tree depth has been reached. However, this approach so far has a tendency to overfit. To counter this, XGBoost utilizes regularization parameters (compare step i) and pruning, meaning cutting back some branches after fully growing the tree, to allow for better generalization and prediction on unseen data. This is done by iteratively performing a bottom-up search and deleting the branches with negative gain scores (v).

Lastly, as in subsection 3.3, each subsequent tree is built by calculating new residuals based on the output of the previous tree. The final prediction of the model aggregates the predictions of each individual tree, denoted as the Output Value.

**Mathematical Background**

Similarity Scores in subsection 3.4 are defined as:

$$S_j = -\frac{1}{2}\frac{\left(\sum_{i\in I_j} g_i\right)^2}{\sum_{i\in I_j} h_i + \lambda}$$

If we utilize the most common Loss Function

$$L(y_i, F(x)) = \frac{1}{2}(y_i - \hat{y}_i)^2$$

and calculate the $g_i$ (gradient, meaning the first derivative) and $h_i$ (hessian, meaning the second derivative), results in:

$$g_i = -(y_i - \hat{y}_i)$$

$$h_i = \frac{d^2}{d\hat{y}_i{}^2}\frac{1}{2}(y_i - \hat{y}_i) = \frac{d}{d\hat{y}_i} - (y_i - \hat{y}_i) = 1$$

$$S_j = \frac{\sum r_i^2}{\sum \mathbb{I}(r_i) + \lambda}$$

which is the sum of residuals in a node squared in respect to the number of residuals and the regularization parameter $\lambda$. To gain intuition, we compare the Figure 15.

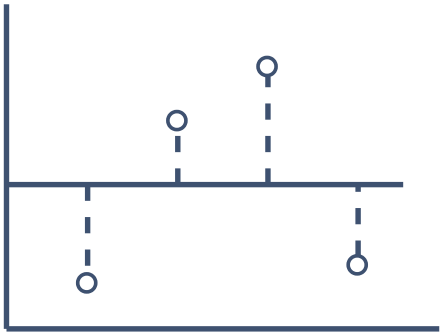


Figure 15: Residual cancellation and node similarity in XGBoost. When a node mixes positive and negative residuals, they cancel out and thus produce lower similarity. Nodes with the same sign (both below or above the horizontal line) give higher similarity [52].

As we see, if data points with opposing effects on the target variable (measured by residuals) fall into the same node, their values cancel each other out, decreasing the numerator and thus lowering the overall similarity score. If the residuals are similar, there is no cancellation, and the similarity score remains relatively large. Furthermore, the $\lambda$ term regularizes the model by reducing the prediction's sensitivity to individual observations, by increasing the denominator and thus lowering the overall similarity for a node as $\lambda$ increases.

The Output Value is very similar to the similarity score, but without the squared sum:

$$w_j^\star = -\frac{\sum_{i \in I_j} g_i}{\sum_{i \in I_j} h_i + \lambda}$$

Furthermore, in pseudo-code, we can define Gain as:

$$Left_{Similarity} + Right_{Similarity} - Root_{Similarity} - complexity\ regularisation$$

This can be mathematically expressed as:

$$\frac{1}{2}\left(\frac{(\sum g_L)^2}{\sum h_L + \lambda} + \frac{(\sum g_R)^2}{\sum h_R + \lambda} - \frac{(\sum g)^2}{\sum h + \lambda}\right) - \gamma$$

The primary use of $\gamma$ is to counteract overfitting during the pruning phase. The one-half factor in front is just for convenience when deriving the formula. This expression quantifies how much better the split is, i.e., how well the leaves cluster the residuals compared to the root.

Once again, we denote the dataset as $\{(x_i, y_i)\}_{i=1}^n$. Furthermore, we define the boosting objective function as:

$$\text{Obj} \approx \Big(\sum_{i \in I_j} g_i\Big) w_j + \tfrac{1}{2}\Big(\sum_{i \in I_j} h_i + \lambda\Big) w_j^2 + \gamma T$$

This is the second-order Taylor approximation, consisting of easy-to-solve sub-terms. We note that $g_i$ is simply the gradient (i.e., the first derivative of the loss function with respect to the predicted value), and $h_i$ is the Hessian (i.e., the second derivative of the loss function with respect to the predicted value).

To calculate the optimal value, we take the derivative with respect to the Output Value $w_j$ and set it equal to zero to solve for $w_j$. We can follow the calculations here:

1. $$\frac{d}{dw_j}\Big(\sum_{i \in I_j} g_i\Big) w_j + \tfrac{1}{2}\Big(\sum_{i \in I_j} h_i + \lambda\Big) w_j^2 + \gamma T = \Big(\sum_{i \in I_j} g_i\Big) + \Big(\sum_{i \in I_j} h_i + \lambda\Big) w_j$$

2. $$\sum_{i \in I_j} g_i + \Big(\sum_{i \in I_j} h_i + \lambda\Big) w_j = 0 \iff w_j = \frac{-\sum_{i \in I_j} g_i}{\sum_{i \in I_j} h_i + \lambda}$$

Next, we plug in the most commonly used loss function, as previously seen in subsection 3.3: $L(y_i, F(x)) = \frac{1}{2}(y_i - \hat{y}_i)^2$.

The first derivative (gradient) is:

$$g_i = -(y_i - \hat{y}_i)$$

Substitution delivers:

$$w_j = \frac{-\sum_{i \in I_j} g_i}{\sum_{i \in I_j} h_i + \lambda} \iff \frac{\sum r_i}{\sum_{i \in I_j} h_i + \lambda}$$

The second derivative (hessian) is:

$$h_i = \frac{d^2}{d\hat{y_i}^2}\frac{1}{2}(y_i - \hat{y}_i) = \frac{d}{d\hat{y}_i} - (y_i - \hat{y}_i) = 1$$

Further substitution results in:

$$w_j = \frac{\sum r_i}{\sum \mathbb{I}(r_i) + \lambda}$$

which is nothing else than the sum of residuals divided by the number of residuals plus regularization term $\lambda$.

In summary, XGBoost is the state-of-the-art machine learning model for tabular data (compare [5]). However, the default implementation assumes that the target data is fully observed. This is not applicable to all real-world scenarios because observations may be incomplete or censored, meaning the outcome is unknown. This was the case in our experiments, where we recorded solver runs until a predefined termination/cutoff time due to the computational limitations of NP-hard problems. This leaves the information partial, meaning that in some instances, we only know the lower bound, but not the true runtime. To address these cases, we introduce survival analysis, a method designed to manage censored data.

## 3.5 Survival Analysis

Survival Time Analysis [41] was first introduced in medicine. The classic example attempts to model, given a diagnosis of an illness in a patient combined with various health statistics, the time to death. However, as such studies can span over decades, many of the subjects may live a considerable amount of time after diagnosis. Survival Time Analysis aims to incorporate this useful information into regression by including samples (subjects) with no recorded death, using the sole information of the lower bound (time alive). The term Survival Analysis is the established name in the literature, but it is also referred to as Time-to-Event Analysis and can be applied beyond time-to-death scenarios. There are different methods for Survival Analysis, such as the Kaplan-Meier estimator, Cox regression, or the Accelerated Failure Time model (AFT), which we introduce below.

#### Kaplan Meier

The Kaplan-Meier estimator [32] is a non-parametric method (meaning it makes no assumptions about the underlying distribution) for estimating the survival function. Consider the Figure 16 below:

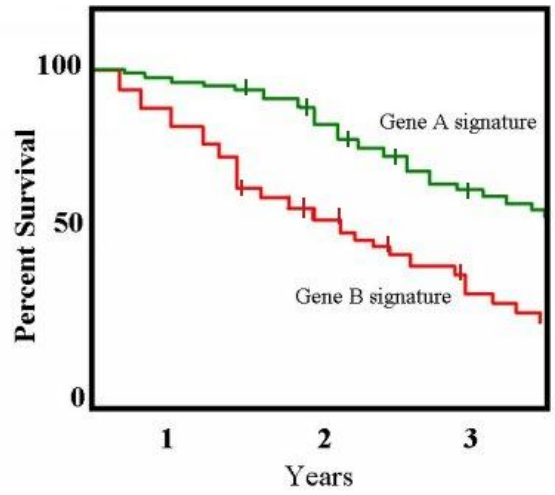



Figure 16: Example of Kaplan-Meier plot for two conditions associated with patient survival. Image from [54].

The Figure 16 shows the following relationship: the X-axis depicts the number of years after diagnosis ($x = 0$ is the diagnosis time). We can now read the probability that a certain subject will survive after diagnosis - for instance, after 2 years ($x = 2$) - for the explanatory variable $y$ = Gene B signature. The probability lies slightly above 50%, meaning we are about 50% confident a patient is still alive after 2 years.

We can model this mathematically with a survival function:

$$S(t) = \prod_{i:t_i \leq t} \left(1 - \frac{d_i}{n_i}\right)$$

With $t_i$ denoting the time when at least one death occurred, $d_i$ the number of these deaths at time $t_i$, and $n_i$ the number of subjects who survived (censored entries) up to time $t_i$. In our case, we would model $x$ as solver runtime in seconds and $y$ as our certainty that a certain instance would terminate within the specified amount of seconds $x$.

Kaplan-Meier requires the following criteria for valid prediction [55]:

1. Censoring is independent: censored time of one subject should not provide any information about censored time of others.
2. Censoring is non-informative: censoring should be random and not correlated with the target-variable.
3. Survival probabilities do not change with time: for instance, prediction is only valid if there are no treatments that can influence survival probability over time.
4. No competing risks: Different other outcomes, such as different cause of death than the one considered in observation, is not allowed.
5. No extrapolation over censored data: KM cannot estimate survival beyond the last occurrence of the uncensored event.
6. Non-parametric: it models only one explanatory-variable at each time (compare two distinct plots for Gene A and B signature Figure 16).

### Cox Regression

Cox Regression (CR) [12] is a semi-parametric function used to determine the impact of multiple independent explanatory variables on survival time. It stems from the motivation that many survival time studies are not interested in predicting the explanatory variable $y$, such as time of death, but in measuring how different explanatory variables influence the said outcome $y$. Furthermore, it can be used to compute continuous survival curves for combinations of explanatory variables, whereas Kaplan-Meier can only handle one explanatory variable at a time, in discrete intervals.

CR is based on hazard functions:

$$h(t|x) = h_0(t) \cdot exp(\beta^T x)$$

which measures the instantaneous rate of death at time $t$, given that the patient has survived up to time $t$ (censored), for explanatory variable $x$. This is different from the survival function $S(t)$, which measures the probability that the event has not happened yet by time $t$. $h_0(t)$ is the baseline hazard, meaning the hazard when all explanatory variables are zero. This parameter can be left unspecified, meaning it makes no assumption about the underlying distribution (this is the non-parametric component of the function). The vector representing learned coefficients, which model the influence of each explanatory variable, is denoted by $\beta$. The term $\exp(\beta^T x)$ adapts the baseline hazard, weighted by the coefficient $\beta$.

The model is trained using partial likelihood [13], which considers the order of event times rather than their exact values. This is why it can be calculated using censored data.

Cox Regression requires the following criteria for valid prediction (expanding on the criteria mentioned for Kaplan-Meier) [56]:

1. Log-Linear Explanatory-Variables: CR requires the effect explanatory-variables have between each other on the hazard-rate to be at most log-linear.
2. Limited Multicollinearity: Non pairs of explanatory-variables should be highly correlated.
3. No outliers: All target-variable values should be tight together.
4. Proportional Hazard Assumptions: each instance pair should have the response to explanatory-variables with regard to hazard-rate.

### Accelerated Failure Time Model

Described by Wei et al., the Accelerated Failure Time Model (AFT) [62] is a common alternative to Cox Regression. While the focus of CR lies mainly in examining how different explanatory variables influence the hazard of the failure time variable, AFT is a simpler model that directly regresses the logarithm of survival time (the target variable) on the explanatory variables, similar to other regression methods (compare subsection 2.3), with the difference that AFT can handle censored data.

The model has the following form [65]:

$$\ln Y \;=\; \langle \mathbf{w}, \mathbf{x} \rangle \;+\; \sigma Z$$

where $\mathbf{x}$ represents the explanatory-variables vector, $\mathbf{w}$ the explanatory-variable coefficients (same as $\beta$ in CR), ln the natural logarithm, and $Y$, $Z$ are random variables representing the target variable and a known probability distribution modeling noise, respectively.

## 3.6 Discussion

We have now provided a theoretical overview of the following machine learning regressors: Regression Trees, Random Forest, Gradient Boosting, and XGBoost. We also introduced the concept of survival analysis, which accurately handles censored runtimes resulting from solver runtime limits. Next, we will compare the aforementioned machine learning architectures and justify the ones that will be evaluated later in this thesis.

Small changes in the training set can result in very different Regression Trees and, consequently, high variance in the predictions. In the context of solver runtime prediction, Hutter et al. [27] found that regression trees were consistently outperformed by Random Forest, which improves high variance with bootstrapping and aggregating techniques. This behavior is highlighted in the literature, which shows the difficulty of balancing a single tree between overfitting and good generalization [25]. Beyond bagging, boosting ensembles improve accuracy further by fitting trees sequentially and learning from previous models errors.

XGBoost, introduced by Chen et al. in 2016 [11], is an open-source boosting system that has achieved the best results in many machine learning (ML) challenges, winning 17 out of 29 competitions on the popular website Kaggle. We found evidence that XGBoost outperforms Random Forest on tabular data similar to that used in our experiments. Fatima et al. (2023) [19] reported up to a 51% improvement in the mean squared error (MSE) loss function for XGBoost over Random Forest in their study. The authors attribute this better performance to the following factors: (1) XGBoost has better overfitting control due to the characteristics of the tree-building process; (2) XGBoost has native handling of missing values; and (3) XGBoost has a strong ability to model complex data through boosting. In another meta-study, Bentejac et al. (2019) [5] compared XGBoost, RF, and Gradient Boosted Trees (GBT) on 28 datasets. GBT won on **10** out of the 28 datasets, with XGBoost close behind in second place with **8** wins.

One goal of this thesis is to improve the machine learning architectures for runtime algorithm predictions presented in the literature, particularly the models and results reported by Hutter et al. For our use case, we believe that XGBoost is the most promising model for the following practical reasons: it offers GPU acceleration and has built-in support for missing values, which is important because the explanatory variables/runtime tables that we use in subsequent sections contain many missing entries. Additionally, the XGBoost library supports more learning objectives, including the accelerated failure time (AFT) survival objective that we will use later in this thesis to handle censored runtimes.

Next, we will establish a baseline for algorithm runtime prediction by analyzing and modernizing the empirical performance models for runtime prediction proposed by Hutter et al. in 2014 [27]. We will use their datasets and explanatory variables for MIP combinatorial optimization algorithms (a subroutine available in ProvideQ), as well as pre-processing and evaluation metrics. The next section will serve as a basis for our contribution to the experiment conducted by Hutter et al., in which we will implement survival analysis with the state-of-the-art XGBoost algorithm on new, modern datasets.

# 4 Algorithm Runtime Prediction: Methods and Evaluation (RQ1)

In 2014, Hutter et al. presented the ”largest empirical analysis of its kind,” thematizing Empirical Performance Models. To establish a baseline for our further contributions, we first study, re-implement, and update their work. This section summarizes what the paper discovered and how we reproduced the results (and what we had to change)Lastly, we discuss the changes Hutter et al. proposed and how this translates into our subsequent survival-analysis and XGBoost contributions.

Before presenting our reproductions and contributions, we briefly introduce the concept of empirical performance models, the foundation of this section.

## 4.1 Empirical Performance Models (EPMs)

Empirical performance models are supervised machine learning (ML) models used for two domains of supervised machine learning - classification or regression. This term was popularized in 2014 by Hutter et al. in their extensive (meta-)study [27], which proposed novel methods and summarized literature dating back to the 1975 study by John R. Rice titled ”The Algorithm Selection Problem”. We depict the pipeline for building EPMs derived from Hutter et al. [27] below in Figure 17.

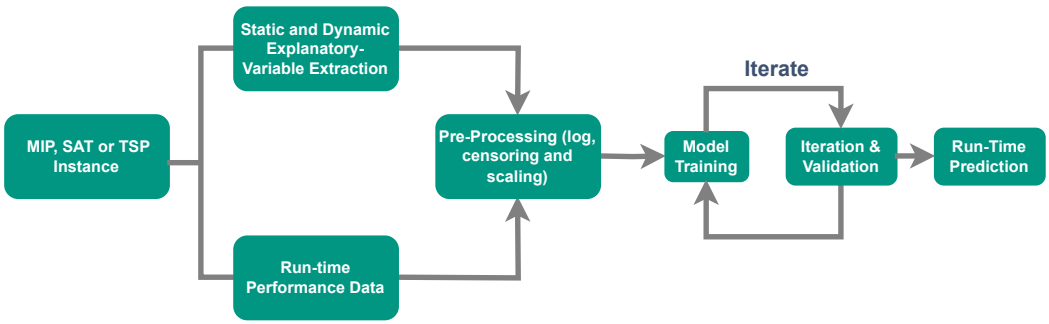



Figure 17: Pipeline used for training Empirical Performance Models (EPM). From problem instance and explanatory-variables extraction, followed by pre-processing and model training, finished with validation, prediction and finally model deployment & application.

In the field of EPMs, explanatory variables for instances such as SAT and MIP are usually structural properties, sizes, and other probing (or pre-solving) statistics. Selecting these variables is an important aspect of EMPs, particularly balancing their calculation time with their effect on the model's overall prediction.

Now, we will compare the main use cases and the empirical performance models for classification and regression. The key difference lies in the form that the predicted values take.

(a) In the first variant, classification, the model takes the explanatory variables and outputs a discrete label. In the domain of empirical performance models, labels can describe problem hardness, for example, solvable or unsolvable, or the proposed solver on a per-instance basis. The second variant is derived from the fact that many

free and commercial solvers are available and differ in performance. For example, well-known MIP solvers, such as Gurobi and CPLEX, as well as the open-source LP-Solve, implement various internal solving techniques and differ in the options that users can set before the solving process begins. This leads to situations in which one instance is solved more quickly and yield better results with one solver than another, and vice versa. Empirical performance model classification enables users to utilize the model to determine which solver could offer the best results for each problem instance.

(b) The second variant is regression, in which the model takes explanatory variables and outputs a continuous label. Instead of labels, the target variables include metrics such as solver runtime (e.g., 300 seconds) or solution quality for approximate algorithms (e.g., a percentage describing how close the solution is to the optimum, e.g., 80%).

When we compare classification and regression, we notice one crucial aspect. Classification provides domain expertise by proposing a label without clarifying why the decision was made, giving it a black-box character. This approach is suitable for many use cases, such as solver selection, where the goal is to achieve the best results without considering possible run times or final solution quality. If modeled properly, regression allows us to make the same decision. However, now the user is empowered to make an informed decision by taking the regressed values into consideration. For example, in MIP solution quality regression, if both commercial CPLEX and open-source LP-Solve are equally good, we may choose the open-source variant. If we had used classification, the user would not have been empowered to make a decision and would have picked the proposed variant, even though they might have preferred the other option given the additional information about the same solution quality.

Based on the argumentation above, we chose to focus on the regression empirical performance models. In the next section, we outline the publication by Hutter et al. [27] that introduced EMPs and focused on run-time regression for traveling salesman problem, boolean satisfiability problem and mixed integer programming.

**Our motivation**

Our goal is to integrate EPMs into ProvideQ for classical and quantum-classical algorithms under fixed solver pre-set solver settings.

The paper "Algorithm Runtime Prediction: Methods & Evaluation" by Hutter et al. [27] is still the largest and most-cited comparison for Empirical Performance Models as of 2025, but it is more than a decade old. Its toolchain is not up to date: the code depends on deprecated MATLAB/C libraries (and does not work out of the box), feature extractors rely on outdated CPLEX APIs, and the datasets are no longer available. Nevertheless, their work remains a classic, extensive, state-of-the-art reference that many publications cite when exploring specific branches of the EMP topic. Our goal in integrating EMPs into ProvideQ explores a new and under-researched use case, which leads us to believe that starting with a foundational publication is beneficial.

For this reason, we restored the legacy code and validated the performance of their best performing model on the relevant ProvideQ MIP vertical slice. Since there is

little standardized benchmarking in this area, our reproduction enables us to establish a clear baseline for reliably comparing our later improvements with XGBoost and survival analysis.

With this scope set, the next subsection will provide a brief recap of the Hutter et al. study in preparation for our reproduction. The detailed results and our contributions to their work will follow.

## 4.2 Introduction To Extensive Meta-Study

The publication by Hutter et al. titled "Algorithm runtime prediction: Methods & evaluation" [27] contains an analysis of 11 ML methods for runtime prediction across 10 datasets for 3 NP-hard problems—SAT, TSP, and MIP—paired with a total of 12 different solvers, such as Gurobi or CPLEX, for each problem.

The following section summarizes the research questions posed by Hutter et al. Their meta-study addresses runtime prediction for three NP-hard problems (SAT, TSP, and MIP), all of which are subroutines available in ProvideQ. However, our summary focuses on MIP because it is the most general modeling problem of the three. Many SAT and TSP problems can be encoded as MIP, and MIP is used the most in real-world applications. As a result, we will use our MIP baseline as a template for implementing Empirical Performance Models for SAT and TSP with minimal changes in future work.

We organize this overview around six main questions from the Hutter et al. study. We focus on the following three aspects from their work: (1) their section 4: new modeling techniques for EPMs, (2) their section 5: problem-specific instance features, and (3) their section 6: performance predictions for new instances.

We also briefly describe all other remaining aspects from their original work: (4) their section 7: performance predictions for new parameter configurations, (5) their section 8: performance predictions in the joint space of instance features and parameter configurations, and (6) their section 9: improved handling of censored run times in random forests. Those aspects are only briefly analyzed because we consider them either not fully applicable to our work or wish to revisit them in further research.

**Models and explanatory-variables.**

2 of the 11 ML methods analyzed were introduced for the first time in the context of runtime prediction by Hutter et al. - namely, Gaussian Processes and Random Forests, the latter proving to be the best ML method for runtime prediction according to their results. Another contribution was the introduction of special explanatory-variable classes: *probing* and *timing/presolving*. Presolving, for example, gives insights into initial solver runtime measured in CPU time for a specific instance on a 5-second CPLEX run. Probing, on the other hand, looks at the output of CPLEX during the 5-second presolve phase to gain a first impression of instance hardness. We note that Hutter et al. used only CPLEX to calculate probing and presolving of MIP explanatory variables, even though they used GUROBI to solve MIPs. We presume this is because CPLEX's logging is more detailed than GUROBI's.

This addition of novel dynamic explanatory variables provided significant improvements over the exclusive use of static explanatory variables (such as instance size),

which had been used previously. In addition to these contributions, their publication classified the explanatory variables based on the time required for their calculation (trivial, moderate, and expensive classes) and evaluated the importance of each class for accurate prediction.

#### Configuration Space

Furthermore, the authors analyzed predicting the runtime of parameterized algorithms. Until this point, they had focused on runtime prediction for default solver settings across many instances. However, solvers are highly configurable - for instance, CPLEX has a parameter space of $1.9 \times 10^{47}$ configurations. In this research question, Hutter et al. reversed the approach from the previous sections. Now, they considered only a single fixed instance, but across 1000 sampled configurations. Since they examined only one instance, they did not calculate explanatory variables from the instance. Instead, they used the 1000 different configurations for that instance, recorded the run-times, and attempted to make runtime predictions for unseen configurations.

#### Joint Instance & Configuration Space

Moreover, Hutter et al. have analyzed the combined impact of runtime-prediction and configuration space: predicting runtime for multiple instances and multiple sampled configurations. They built pairs of (instance, config), once again including instance explanatory variables. The results showed that RF was the best or tied for best on every dataset. Furthermore, the predictive error decreased slightly compared to using only instance-based runtime prediction.

#### Survival Analysis

Lastly, Hutter et al. noticed in their analysis of EPMs for solver runtime prediction one significant flaw. For each instance, solver run-times were capped at a time limit of 3600 seconds, making them "censored." Right-censored run-times mean that we know the run lasted at least up to a cap time $k$, but not how long it would take beyond $k$. Treating each censored run as if it finished in k seconds introduces a strong downward bias into the model, which leads to significant underestimates on instances that could potentially take much longer to solve. One might interpret a model prediction near the $k$ value as indicating "possibly very long runtime," but it is impossible to differentiate between true runs that are close to k and those that would take significantly more time.

This motivated Hutter et al. to implement survival analysis on the joint instance & configuration space runtime prediction mentioned in the paragraph above. They analyzed an implementation by Schmee & Hahn, which used the expectation maximization (EM) algorithm for linear models handling censored data points. They adapted this idea to RF by using the EM algorithm to iteratively impute and update the predictions for the censored samples. The reported findings considered two cases: (a) survival analysis with a fixed k per experiment, and (b) instance-specific k, defined as the "best-config-time", determined by sampling run-times for the same instance on different configurations. The results showed that survival analysis with fixed k, as used in Sections 1–6, slightly improves loss function measured in Root

Mean Square Error (**RMSE**) but is reliable only up to $2 \times k$ (as indicated by a poor log-likelihood (**LL**) score) because a single k limits available information. Results for instance-specific k delivered better **RMSE** and higher certainty as measured by **LL**.

Now we will argue why we focus on the pure solver runtime prediction without configurations and survival analysis.

#### Rationale For Our Focus

From this point on, as mentioned we focus only on the first case described by Hutter et al. - instance runtime prediction without configuration of solvers. The reasoning is that in the ProvideQ framework, the primary need is fast runtime prediction using pre-selected solver settings. Furthermore, calculating runtimes for more than 1000 instances with a 1-hour cut-off cap proved to be resource-intensive enough. Evaluating 1000 configurations per instance would require $10^8$ CPU-seconds, which would be beyond the scope of this bachelor's thesis. Additionally, configuration sampling combined with runtime prediction is more naturally used as training data for a multi-label classifier, where the goal is to predict the best configuration to maximize the chance of solving a certain instance, not necessarily to predict runtime.

Therefore, from this point onward, we focus on the simpler variant from Sections 1–6 described in [27], namely runtime prediction for new instances using fixed solver defaults. For this reason, we will also not re-implement the survival analysis modeling proposed in Section 9 in this baseline reproduction, but we will research and implement survival analysis using XGBoost and the new dataset in further section Research Question 2.

We begin with a detailed dive into first crucial aspect of their work: the MIP datasets Hutter et al. have used for their ML model training and evaluation.

## 4.3 Detailed Insights: Data, Models, Results

#### Datasets

The available MIP sets utilized in [27] are as follows:

Table 1: MIP datasets considered in the reproduction study.

| Set | #inst. | Domain | Used? |
|---|---|---|---|
| BIGMIX | 1510 | Mixed real/synthetic | main baseline |
| CORLAT | 2000 | Wildlife corridors | (not re-run) |
| RCW | 1980 | Wildlife sustainability | (not re-run) |
| REG | 2000 | Synthethic auction winner determination | (not re-run) |
| CR | 3980 | CORLAT ∪ RCW | (not re-run) |
| CRR | 5980 | CORLAT ∪ RCW ∪ REG | (not re-run) |

BIGMIX is a large and diverse collection of 1510 MILP benchmarks composed by Hutter et al. This dataset was chosen for its high heterogeneity, averaging 8610 variables and 4250 constraints, with a maximum of 550,539 variables and 550,339

constraints. CORLAT is a homogeneous dataset containing real-world MILP instances from wildlife corridor planning for grizzly bears in the Northern Rockies region [23]. RCW comes from a computational sustainability study on protecting parcels to slow the spread of the endangered red-cockaded woodpecker. RCW is synthetic and was generated using the generator provided by [1]. REG contains synthetic instances from the winner determination problem in combinatorial auctions, generated using the Combinatorial Auction Test Suite provided by [39]. Lastly, CR and CRR are composite sets created to generate a more heterogeneous dataset from highly homogeneous ones.

Unfortunately, exact reproduction of the datasets mentioned above was not possible. The code and data provided in the paper included only precomputed explanatory-variable and runtime tables for all the instances and solvers, but not the instances themselves. At first, we attempted to reproduce the instances, but the links provided in the referenced work to these datasets are either invalid or, as in the case of RCW and REG, point only to the instance generators, not the instances themselves. This poses problems, as these generators are more than a decade old (and do not run out of the box) and furthermore rely on seeding. The time we would need to invest in modernization, combined with the fact that we still could not reproduce the exact instances used in the work by Hutter et al., led us to opt for using the precomputed tables provided, as this does not hinder our goal of experiment reproduction.

As we wish to utilize this work as a baseline, among the available MIP sets, we focused on BIGMIX, because, as the authors themselves state, it is the most heterogeneous dataset and thus allows for the best generalization, which is exactly what we need from a baseline model.

Now that we have an established dataset, we will explain the explanatory variables that Hutter et al. derived from the instances in the MIP-BIGMIX dataset, which was proposed for runtime prediction in their work.

**Explanatory-Variables**

MIP instances are encoded as large coefficient matrices. The runtime is mainly determined by how long the branch-and-cut search takes (compare MIP). The success of EPMs highly depends on how well the explanatory variables capture the hardness of the MIP instance with regard to solver runtime. The calculation of explanatory variables can be costly. One main concern is cases where the calculation of the explanatory variables takes longer than the solving process itself. For this reason, balancing predictive accuracy with the cost of explanatory-variable calculation must be taken into consideration.

Hutter et al. introduced, through experimentation and domain insights, a total of 121 explanatory variables for MIP. They differentiate between static and dynamic variables while also categorizing the cost of calculation into cheap, moderate (linear), and expensive (polynomial). Static variables, which are cheap or moderate to calculate, are available without running any solver - e.g., number of variables/constraints. They are always a good and safe baseline because, as they are directly calculated from the instance itself, they model the input space well. However, they prove to be insufficient on heterogeneous sets [27]. Dynamic variables are based on probing and timing. Hutter et al. construct the probing variables by briefly calling

CPLEX for 5 seconds and observing the algorithm's trajectory, which is similar to meta-learning. Timing features measure the time required to compute other groups of features (besides timing/probing). The main issue with probing variables is the overhead they introduce - typical ML predictions take no longer than a few milliseconds. Furthermore, as stated in [26], all MIP solvers incorporate seeding and heuristics into their solving processes, introducing high variability in solving time and making reproducibility difficult.

All of the explanatory-variables were calculated on 55 × dual-core 3.2 GHz Intel Xeon cluster with 2GB of ram, running OpenSuSE 11.

The explanatory variables introduced by Hutter et al. are as follows:

Table 2: Feature groups used by Hutter et al.

| **Group** | **ID range** | **#** | **Brief description** |
|---|---|---|---|
| Problem Size & Type | 1–25 | 25 | Basic metrics (e.g., # of vars) |
| Variable–Constraint Graph | 26–49 | 24 | Bipartite-graph stats |
| Linear Constraint Matrix | 50–73 | 24 | Sparsity, medians, variances |
| Objective Function | 74–91 | 18 | Sum, mean, median of objective coeff. |
| LP-based | 92–95 | 4 | LP objective value, graph metrics |
| Right-Hand Side (RHS) | 96–101 | 6 | Statistics of RHS values |
| MIP Probing (new) | 102–116 | 15 | 5s CPLEX presolve stats, cut counts |
| Timing (new) | 117–121 | 5 | CPU time to compute feature groups |

Costs for variables in the ranges (1–25, 96–101) are trivial, (26–91) are cheap, (92–95) are expensive, and all probing and timing features are moderate to calculate. The difference in predictive performance (measured by **RMSE**, where lower is better) can be seen below:

Table 3: Performance of CPLEX on BIGMIX.

| | Alg. run [s] | | RMSE | | | | | Avg. feat. $t$ [s] | | | | Max. feat. $t$ [s] | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Scenario | avg | max | triv | prev | chp | mod | exp | prev | chp | mod | exp | prev | chp | mod | exp |
| CPLEX–BIGMIX | 719 | 3600 | 0.96 | 0.84 | 0.85 | **0.63** | 0.64 | 17 | **0.13** | 6.7 | 23 | 1e4 | **6.6** | 54 | 1e4 |

As we can see, there is a high performance gain of about 26% between using only trivial and cheap static features and including moderate dynamic features. Also, LP-based (static) features are expensive to calculate but do not yield significant improvements in prediction quality.

Besides experiment reproduction, we kept all 121 variables for our baseline so that we can isolate the model improvements (from Random Forest to XGBoost). Before feeding these 121 variables into the ML models, Hutter et al. applied pre-processing, which we also replicate exactly. We cover the details in the next subsection.

**Pre-processing**

As preprocessing, Hutter et al. applied z-scoring (standard score) to each explanatory variable. This standardization technique subtracts the mean and divides each value

by the standard deviation, compressing and scaling the entire distribution to match the ideal normal curve, as depicted below.

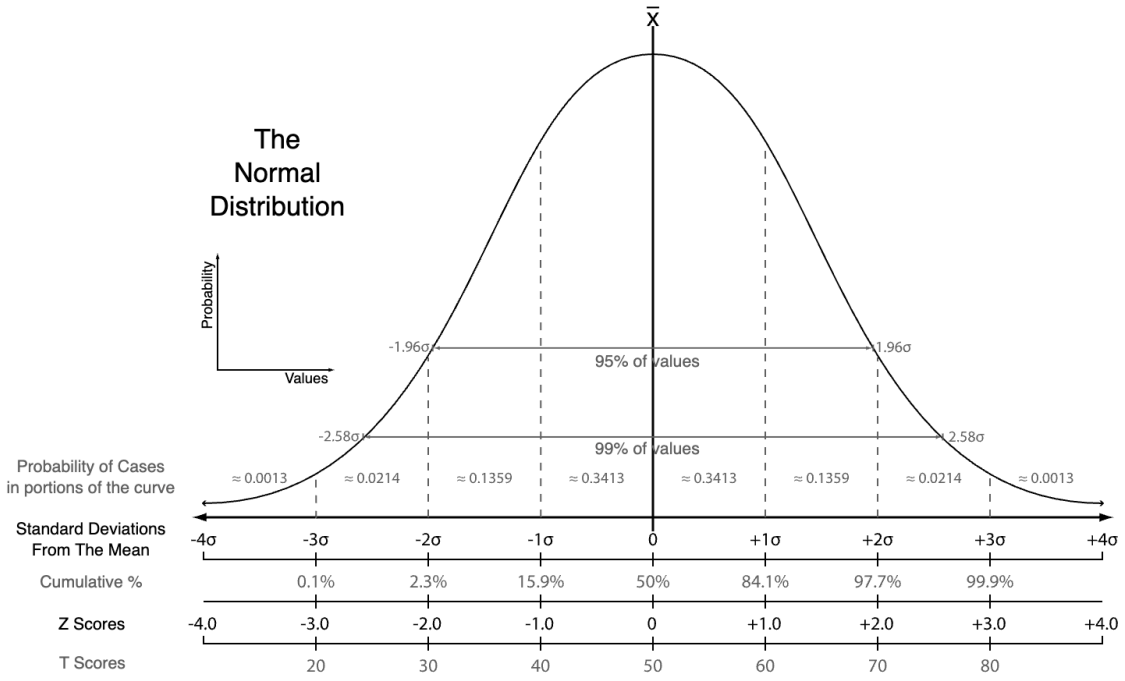



Figure 18: Normal Distribution that illustrates z-scoring: variables are standardized to have zero mean and one unit variance. Image from [63].

This prevents different groups of explanatory variables from being on different scales of magnitude. For instance, the number of variables is in the range of $[10^3; 10^5]$, while RHS values are in $[10^{-5}; 10^{-1}]$. This would cause distance-based ML models to favor explanatory variables with high magnitude while discarding others. However, tree-based ML models consider only which side of a threshold a value falls on (i.e., order) at each split, rather than the magnitude of the value, which is important for distance based-based models.

However, this doesn't hold for the target variables: solver run-times. As an NP-hard problem, by definition, many MIP instances could possibly not be solved within a realistic time frame. To balance hardware resources and potentially large datasets, Hutter et al. implemented a cut-off time of 3600 seconds for solvers. Instances that did not terminate by the cut-off are marked as if they did, with the cut-off written as the solve time. Nevertheless, many simple instances and solver heuristics allow some instances to be solved quickly. In the BIGMIX dataset with the CPLEX solver, the majority of solve times lie in milliseconds or seconds. However, a considerable number of samples are marked with the 3600-second cut-off. This makes the runtimes skewed - the distribution has a sizable heavy tail, forcing the model to fit these outliers while performing worse on the majority due to the high squared-error loss. Hutter et al. applied a logarithmic transformation to run-times $y$:

$$y^* = log_{10}(y + \epsilon) \quad with\ epsilon = 10^{-6}$$

After this transform, the residuals are much more uniform, allowing for better precision due to standardized loss function measured in Root Mean Square Error

(**RMSE**). The log transform has further advantages: after log transformation, the **RMSE** can be more intuitively understood, as the prediction error $\delta = \hat{y}^* - y^*$ of 0 means exact prediction; 1 means one order of magnitude error (×10), ; 2 means two orders of magnitude (×100). The constant $\epsilon$ is added for instances that solved almost instantly, leading to underflow and thus a runtime $y$ saved as 0 seconds, making the log undefined.

All the runtimes were calculated on 55 × dual-core 3.2 GHz Intel Xeon cluster with 2GB of ram, running OpenSuSE 11.

We have now defined the transformed explanatory variables using z-scoring and a log transformation of run times. In the next subsection, we discuss the choice of machine learning (ML) models trained on this preprocessed data. First, we will provide an overview of the models researched by Hutter et al., then motivating our focus on the model with the best performance: Random Forests.

**Models**

Hutter et al. have compared 11 ML-models. We first introduce a slice of the results presented in their evaluation, to prove random forests are not cherry-picked. A subsection of the results (without SAT and TSP solvers) is depicted in Table 4.

Table 4: Predictive error (RMSE) and model-training time for six learning methods on selected benchmark scenarios. Best values per row in **bold**.

| Scenario | **RMSE** | | | | | | **Time to learn model (s)** | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | RR | SP | NN | PP | RT | RF | RR | SP | NN | PP | RT | RF |
| CPLEX–BIGMIX | 2.7E8 | 0.93 | 1.02 | 1.00 | 0.85 | **0.64** | 3.39 | 8.27 | 4.75 | 41.25 | 5.33 | 3.54 |
| Gurobi–BIGMIX | 1.51 | 1.23 | 1.41 | 1.26 | 1.43 | **1.17** | 3.35 | 5.12 | 4.55 | 40.72 | 5.45 | 3.69 |
| SCIP–BIGMIX | 4.5E6 | 0.88 | 0.86 | 0.91 | 0.72 | **0.57** | 3.43 | 5.35 | 4.48 | 39.51 | 5.08 | 3.75 |
| lp_solve–BIGMIX | 1.10 | 0.90 | 0.68 | 1.07 | 0.63 | **0.50** | 3.35 | 4.68 | 4.62 | 43.27 | 2.76 | 4.92 |
| CPLEX–CORLAT | 0.49 | 0.52 | 0.53 | **0.46** | 0.62 | **0.47** | 3.19 | 7.64 | 5.50 | 27.54 | 4.77 | 3.40 |
| Gurobi–CORLAT | **0.38** | 0.44 | 0.41 | **0.37** | 0.51 | **0.38** | 3.21 | 5.23 | 5.52 | 28.58 | 4.71 | 3.31 |
| SCIP–CORLAT | 0.39 | 0.41 | 0.42 | **0.37** | 0.50 | **0.38** | 3.20 | 7.96 | 5.52 | 26.89 | 5.12 | 3.52 |
| lp_solve–CORLAT | 0.44 | 0.48 | 0.44 | 0.45 | 0.54 | **0.41** | 3.25 | 5.06 | 5.49 | 31.50 | 2.63 | 4.42 |
| CPLEX–RCW | 0.25 | 0.29 | 0.10 | 0.03 | 0.05 | **0.02** | 3.11 | 7.53 | 5.25 | 25.84 | 4.81 | 2.66 |
| CPLEX–REG | **0.38** | 0.39 | 0.44 | 0.38 | 0.54 | 0.42 | 3.10 | 6.48 | 5.28 | 24.95 | 4.56 | 3.65 |
| CPLEX–CR | 0.46 | 0.58 | 0.46 | **0.43** | 0.58 | 0.45 | 4.25 | 11.86 | 11.19 | 29.92 | 11.44 | 8.35 |
| CPLEX–CRR | 0.44 | 0.54 | 0.42 | **0.37** | 0.47 | **0.36** | 5.40 | 18.43 | 17.34 | 35.30 | 20.36 | 13.19 |

The abbreviations are as follows: Ridge Regression (RR), Sparse Polynomial Regression with Forward-Backward selection (SP), Neural Networks (NN), Projected-Process Gaussian Process (PP), Regression Tree (RT), **Random Forest (RF)**. The models take as input the already standardized explanatory variables and the log-scaled target variable, as described in the previous section.

The findings show that simpler linear or kernel methods can slightly keep up on homogeneous datasets. However, as soon as they are applied to BIGMIX, which is the most heterogeneous dataset contained in the study, the models can not keep

up. The novel models used in this paper, PP and RF, performed similarly on the homogeneous datasets, but RF yielded about 20-30% better **RMSE** results than all other models on the most diverse dataset, BIGMIX.

Because the Random Forest model consistently achieved the lowest **RMSE** and was fast to train, we will examine it in detail in the next subsection, including the changes implemented by Hutter et al.

#### Random Forest

From this point, we focus on and consider only the best-performing model, Random Forest, which proved to be the most accurate on every MIP set in Hutter et al., with its non-ensemble implementation, Regression Trees, in second place. RF solves many of the problems posed by RT, such as high variance and overfitting, and sensitivity to changes in the dataset, while retaining many strengths, such as low training cost, natural handling of discrete and continuous variables, and good predictive performance.

#### Parameters Description

The listing below describes the hyperparameters and their values from the Hutter et al. implementation of RF.

- **Number of trees** `B` = 10 - they control variance by averaging.
- **Feature-subsampling fraction** `perc` = 0.5 - fraction of variables considered for split at each node.
- **Minimum node size** $n_{\min}$ = 5 - defines how many samples in node are needed to stop splitting.
- **Pruning**: none - variance is reduced by the forest average.
- **Bootstrapping / bagging** - adapted, compare below.
- **Node split-point policy** - adapted, compare below.

#### Hutters et al. deviations from default RF

Hutter et al.'s implementation of Random Forests deviates slightly from those established in the literature. For one, as mentioned in subsection 3.2, RF reduces variance and overfitting on the data through the implementation of randomness. The two main sources are bootstrapping - training each tree on a subset of the dataset - and considering only a random fraction of explanatory variables at each node split (`perc`). Their implementation uses only the latter, as it seemed to perform slightly better during experimentation. We argue that this may be due to the small dataset (for example, 1510 instances in BIGMIX) and thus low sampling noise, as well as the dominance of heterogeneity in the explanatory variables - reducing the data available for each tree hurts the bias more than it helps the variance.

Furthermore, default Random Forest implementations such as the ones described in [25] and [46] do not capture quantitative uncertainty, meaning a forest returns

only the mean of tree predictions. Hutter et al. modified this by additionally storing the leaf-level variance. When given a new MIP instance, a single tree traverses down to a leaf. This leaf memorizes $\mu_\beta$, which is the average log-runtime for the training points (default behavior), and $\sigma^2_\beta$, which is the spread of those runtimes, capturing how noisy these points are (modified). Overall, the runtime is thus predicted with $\mu = \frac{1}{B}\sum_{b=1}^{B} \mu_b$, meaning asking every tree for its best guess and taking the average. Uncertainty comes from two places: (a) model variance, meaning the measure of how much each tree disagrees with the others, and (b) data variance, measuring how an individual tree captured internal noise in its leaves. This variance is captured with:

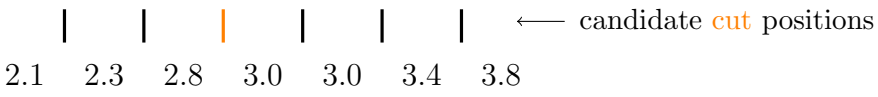
$$\sigma^2 = \underbrace{\frac{1}{B}\sum_{b=1}^{B} \sigma_b^2}_{\text{average within-leaf variance}} + \underbrace{\frac{1}{B}\sum_{b=1}^{B} \mu_b^2 - \mu^2}_{\text{variance of the trees means}}$$

The last change lies in the behavior of node splitting. The default greedy algorithm, as described in Figure 3.1, would follow the steps below.

Imagine a single input feature $x$ with its values sorted in ascending order:

$$2.1 \quad 2.3 \quad 2.8 \quad 3.0 \quad 3.0 \quad 3.4 \quad 3.8$$

The greedy algorithm would try every gap between successive values and pick the one that minimizes the squared error loss (in our case, the gap between 2.8 and 3.0):




Then, the threshold for the split would be placed exactly in the middle, meaning

$$s = \frac{(2.8 + 3.0)}{2} = 2.9$$

However, this means that no matter how many trees are in the ensemble, every tree that selects explanatory variable $x$ at a certain split point would pick the same $s$. Hutter et al. kept the best gap - in our case, the one between 2.8 and 3.0 - however, instead of placing the split at the midpoint (2.9), they drew the exact split uniformly at random from $s \sim \mathcal{U}(2.8,\ 3.0)$. This allows for smoother mean predictions and introduces additional uncertainty, which can further help reduce variance.

Lastly, current implementations set the default number of trees in the forest to $B = 100$, while Hutter et al., due to hardware limitations a decade ago, used a smaller value of $B = 10$.

Now that the changes to the RF variant proposed by the authors have been specified, we will answer the following detailed evaluation questions: What metrics were used?

### Evaluation Metrics

Hutter et al. reported the performance of their RF model using 10-fold cross-validation (Figure 2.3) without a stratification technique, meaning the data was randomly scattered across each fold. All the pre-processing described in Table 4.3 was done inside each training fold and then applied to the hold-out fold to avoid information leakage.

### Error metrics

All the error-metrics has been computed on the $log_{10}$ transformed run-times.

1. **RMSE** = $\sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \mu_i)^2}$
2. **Pearson Correlation** (**CC**) = $\frac{\sum_i(\mu_i-\bar{\mu})(y_i-\bar{y})}{(n-1)\,s_\mu s_y}$
3. **Log-likelihood** (**LL**) = $\sum_{i=1}^{n}\log\phi\left(\frac{y_i-\mu_i}{\sigma_i}\right)$

**RMSE** (1) should be interpreted as lower equals better. It captures the difference between observed (true) log-runtimes and the model's predicted mean for the same instance.

**CC** (2) measures how well the predictions and true runtimes align with a straight line. If the predicted and true runtimes align, **CC** is near 1. If they do not align and show high variance, **CC** is near 0. **CC** is not a ranking metric; it rewards linear relationships. If every prediction is off by a constant factor, such as $\times 1.5$, **CC** can still be close to 1, however it drops when the relationship is noisy.

**LL** (3) is only possible to calculate because of the variance inclusion in Hutter et al.'s implementation. **LL** measures the uncertainty of the predictions by rewarding accurate means and calibrated uncertainty.

With understanding of the evaluation metrics, we will next verify if the reported results reproduce the original findings. We will execute both the MATLAB legacy code and our modernized MATLAB version on the same data and report **RMSE**, correlation and log-likelihood. The following subsection utilizes both tabular and visual representations of the results.

### Reproduction: Ours vs. Paper

First, we begin by reproducing the experiment using the code provided by Hutter et al. The goal is to verify whether our attempt to run the legacy code in the year 2025 worked as expected and whether we can obtain results similar to those presented in the original paper. The table presented below is an excerpt of the results published in the original publication regarding the **RMSE** scores for all the models and the CPLEX solver on the BIGMIX dataset. The last column presents the results produced by our modernized version of the code library. As mentioned in Figure 4.3, we limit the comparison to a small vertical slice, which is all we need for a baseline.

Table 5: Predictive error (RMSE). Columns 2–7 are the original numbers reported by Hutter et al. the last column is our reproduced RF result. Best value per row in **bold**.

| Scenario | **RMSE (Hutter et al., 2014)** | | | | | | **RF (repro.)** |
|---|---|---|---|---|---|---|---|
| | RR | SP | NN | PP | RT | RF | RF |
| CPLEX–BIGMIX | 2.7E8 | 0.93 | 1.02 | 1.00 | 0.85 | 0.64 | **0.62** |

The results are surprising. Even though we used the exact same dataset, seed, number of folds, and (functional) code, we observed an improvement of about 3.1% in the **RMSE** for Random Forest. This trend continues across other metrics, as depicted below.

Table 6: Spearman rank correlation and log-likelihood on CPLEX–BIGMIX. Columns 2–9 reproduce Hutter et al. (2014); the last two columns are our reproduced Random-Forest (RF) results. Higher values are better. Results which should be compared are color-coded.

| Scenario | **Spearman (Hutter et al.)** | | | | | | **Log-likelihood (Hutter et al.)** | | **RF (reproduced)** | |
|---|---|---|---|---|---|---|---|---|---|---|
| | RR | SP | NN | PP | RT | $RF_{orig}$ | PP | $RF_{orig}$ | Spearman | Log-LL |
| CPLEX–BIGMIX | 0.82 | 0.81 | 0.81 | 0.76 | 0.84 | 0.90 | −8.06 | −0.70 | 0.91 | −0.63 |

We achieved better results for all metrics. Spearman rank correlation in our run shows an improvement of 1.1%, and log-likelihood shows a significant improvement of 8.7%. Although we are unsure, we attribute this difference to the following factors:

1. Random Seed. Hutter et al. could have used for the benchmarks a different seed, than the provided ”1234” distributed with their MATLAB code.

2. Library Rebuilds. In order to run their legacy-code, we had to recompile multiple C and C++ libraries for the Apple-silicon Macs (aarch64-apple-darwin). This has also upgraded the versions of the said libraries in the process - which may have boosted the performance. Exact details can be found in subsection 4.4.

Lastly, we wish to show the visualized RF results on the BIGMIX dataset with the CPLEX solver, as published by Hutter et al.

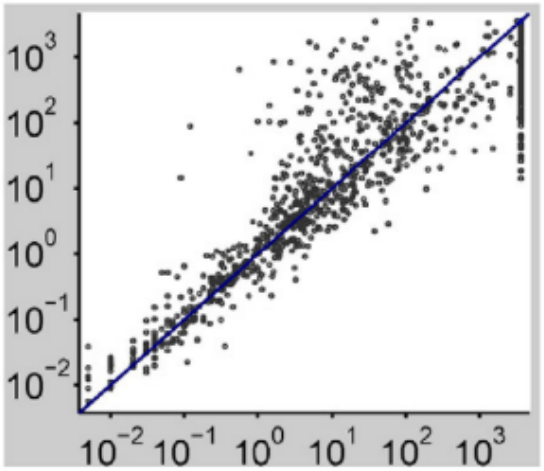



Figure 19: Random-forest prediction (y-axis) vs. actual runtimes (x-axis) on the CPLEX–BIGMIX test set (log-scale; blue line = perfect prediction) [27].

We see that many of the points lie within one order of magnitude error (×10). The calculated **RMSE** in our reproduction was 0.62 on the log scale, meaning the model makes an average error of a factor of 4.17. We believe this margin of error is acceptable for gaining real insights when using the ProvideQ for MIP runtime prediction. This enables users to make an informed choice about which available MIP solver could terminate in an adequate amount of time. However, we notice that the extremes at the tail, specifically at the 3600-second mark, can reach optimistic errors of two orders of magnitude. This is concerning if we recall that the recorded 3600 seconds are not true run-times, but rather right-censored runs that did not terminate within the given time limit. This optimism results from treating censored data runs as exact, and has been discussed in Section 9 of Hutter et al.'s survival analysis.

## 4.4 Technical

In this section of the thesis, we transition from the results to the technical work required to reproduce the experiments on modern hardware. The original code, written for pre-2014 MATLAB, C, and legacy CPLEX API, does not run out of the box. We modernized the code infrastructure without altering the modeling logic. Our updates consisted of cleaning the MATLAB project, recompiling binaries, and fixing the plotting and export logic for figures. Additionally, we updated the MIP explanatory-variable extractor provided by Hutter et al. to the current CPLEX (22.1) and the arm64 processor architecture. We have documented all the changes to enable further researchers to make exact reproductions.

### Framework Adaptation

### Repository

Hutter et al. provided their source code as a ZIP file on the website of the University of British Columbia[1]. First, we wrapped this pre-2014 codebase in a modern MATLAB project so that it can be opened in the modern MATLAB IDE as-is, without the need for manual "addpath" command. We added a startup.m file, which

[1] https://www.cs.ubc.ca/labs/algorithms/Projects/EPMs/

automatically detects the project root, appends every subfolder to the path, sets the Ghostscript executable so that vector PDF export works, and lastly, provides one-liner commands for recompilation of the C/MEX files for Apple Silicon Macs (aarch64-apple-darwin).

### Code and MEX

A MEX (MATLAB Executable) binary is a shared library file, compiled in C, C++, or Fortran, that MATLAB can load and call just like a native MATLAB function. These MEX library files must be compiled for each platform, e.g., Windows, Linux, macOS Intel, and Apple Silicon. The available libraries were for x86 only. Through trial and error, we recompiled the C libraries needed for the custom regression tree code using legacy 32-bit array APIs, with maximum optimization enabled and targeting the 1999 C language standard. We then recompiled these updated C libraries into Apple Silicon-compatible "mexmaca64" binaries.

As mentioned above, we included the commands used for this compilation so that users on other platforms (e.g., Linux) can do the same with minimal overhead. Furthermore, we removed certain memory calls such as pack, which were removed from modern MATLAB environments to prevent slowdowns and crashes in the 64-bit version of MATLAB.

### Code Entry-Point

We cleaned the main experiment script (do_ehm_experiments.m) by removing about 200 lines of dead or debug code. We replaced variables with generic names such as k with more descriptive names for easier code interpretation. We also refactored and moved to the main file some unintuitive boolean flags buried deep in the code, which determined important experiment parameters.

### Plotting

We updated the plot generation to use MATLAB's built-in exporter instead of the deprecated and non-functional "export-fig" library.

### Documentation

We uploaded our modernized code to GitHub[2] and updated the *README.md* with a quick-start guide. We also added a *CHANGELOG.md* listing every change made relative to the 2014 release.

### Note

We focused only on the code needed to reproduce the runtime prediction experiment for new instances with fixed solver defaults. We chose not to remove code related to other models and experiments from [27], so that future researchers interested in reproducing (or modernizing) other parts of the code do not have to reconstruct the original source.

[2] https://github.com/tubadzin/Performance-Prediction-for-Subroutines-in-Meta-Solver-Strategies

#### Explanatory-Variable Extractor

#### Repository

Hutter et al. also provided their source code for feature calculation on their website[3]. At the top level, we removed 11 files that were dead code and artifacts (such as examples, logs, and backup binaries) left over from experimentation. We created a new README.md with added documentation and a sample input, adapted the Makefile, and included a test instance. Lastly, we built a ready-to-use static universal binary of MIPFeature, which can be used directly. We link this modernized repository on GitHub[4].

#### macOS (Apple-silicon) port

We switched the target in the Makefile from the legacy x8_RHEL3.0_3.2 to arm64.osx. For this, we used clang++, which now compiles the C/C++ code with AppleClang, the officially supported toolchain for Apple Silicon. Furthermore, for recompilation, we linked against the CPLEX 22.1 static libraries instead of CPLEX 9.1.

#### CPLEX 22.1 API modernization

Several functions from CPLEX 9.1 were deprecated, meaning the code did not run when we migrated to the newest version, 22.1. We replaced these calls (CPXfopen, CPXgetclqcnt, ...) with their current equivalents, which we identified using the CPLEX documentation.

#### Bug-fixes

We found a numerical bug where, due to missing parentheses, a result was interpreted as a boolean expression instead of an absolute value:

```
fabs(rhs_b[i] > 1e-6) → fabs(rhs_b[i]) > 1e-6
```

This bug caused all entries for the explanatory variable ”rhs” to contain only 0s and 1s, which is not the intended behavior.

Furthermore, as noted by the author Frank Hutter, we detected memory leaks. We did not address them, as the calculator is called per instance, making them manageable.

#### Documentation

We updated the README.md, which walks users through the steps from git clone to feature computation.

#### Note

We have modernized the Feature Extractor code; however, we did not utilize it in this section, as we used the precompiled explanatory-variable tables provided by Hutter et al. We will use this extractor in the next RQ on new, modern datasets.

[3] https://www.cs.ubc.ca/labs/algorithms/Projects/EPMs/

[4] https://github.com/tubadzin/Performance-Prediction-for-Subroutines-in-Meta-Solver-Strategies

## 4.5 Discussion

### Key Findings

Overall, we have analyzed, reproduced, and validated the 2014 results from Hutter et al. [27] on a small vertical slice using Random Forest with the (BIGMIX, CPLEX) benchmark.

We find that Random Forest outperformed all other models on the BIGMIX dataset with the CPLEX solver. Surprisingly, our modernized RF reduced **log-RMSE** from 0.64 to 0.62 (an improvement of 3.1%). We found other substantial improvements in the Spearman $p$ score and log-likelihood, of 1% and 9% respectively. We improved the results even though we made no changes to the algorithmic logic - we only modernized the source code and compiler flags. We suspect that our numbers outperform the original likely due to differences in seeding or sample composition in the cross-validation folds. Other possible explanations include changes in the newly compiled MEX libraries used for the custom computation of regression trees.

We further learned that the inclusion of timing and probing features pays off. They deliver a 26% **RMSE** gain for about 5 seconds of predictive overhead. Lastly, we learn that right-censoring matters. As seen in Figure 19, treating censored run-times as actual run-times produces a large, optimistic tail, which underestimates runs that may not terminate.

### Insights for ProvideQ

We find a few practical insights for ProvideQ. A **log-RMSE** of 0.62 means a 4× multiplicative error on average. A practical example: if the RF model presented by Hutter et al. predicts 90 seconds, the true interval is roughly between 23 seconds and 360 seconds. We argue that this could offer good-enough guidance for deployment.

We note that even though BIGMIX was the most heterogeneous dataset tested by Hutter et al., it is not heterogeneous enough by modern standards (compare MIPLIB). We take the results of a 4× multiplicative error on average with a grain of salt and will address this in RQ2 with modern datasets.

Lastly, as Koch et al. [37] noted, modern solvers can reach up to 3 orders of magnitude (×1000) speed-ups. Hutter et al. ran their experiments on the 2009 version of CPLEX. We believe it is necessary to use a modern solver for both runtime and explanatory-variable calculations.

### Possible Validation Errors

We noticed, however, a few possible validation problems. We could not obtain the original datasets and instances used in the paper. Using the precomputed explanatory-variable tables may have bypassed the bugs we discovered in Explanatory Variable Extractor section - we could not validate whether the pre-processing Hutter et al. applied would match ours.

In the next Research Question 2, with a verified baseline, we will explore the improvements we can make to these findings.

# 5 Modernizing Empirical Performance Models (RQ2)

In RQ1 section 4, we showed through replication of the Hutter et al.'s (2014) experiment that Random-Forest (RF) model outperformed 10 other models on the heterogeneous BIGMIX-CPLEX benchmark. The RF model, built in legacy MATLAB, achieved a log-RMSE of 0.64, which translated to the averaged absolute multiplicative error of 4×.

However, (i) MATLAB is rarely used for machine learning (ML) today, (ii) new heterogeneous MIP datasets, such as MIPLIB, have emerged since the 2014 publication, (iii) the new default for tree libraries is the gradient boosted trees strategy, (iv) the experiment was produced using an outdated CPLEX version, and (v) newer hardware and optimization methods, such as Bayesian hyperparameter tuning, are available.

In this section, our goal is to improve the predictive performance of the MIP subroutine runtime prediction in empirical performance models, so that the machine learning model implemented in ProvideQ reflects the current state of the art.

First, we will reimplement the modified RF proposed by Hutter et al. in Python with SciKit-learn, using the same dataset and explanatory variables as in RQ1. The changes made by Hutter et al., particularly the modifications to the node-splitting behavior, are not included in any SciKit-learn framework. Therefore, we had to write a wrapper to add this custom behavior. Then, we will compare the results of the bare reimplementation with those of the same reimplementation with tuned hyperparameters using the modern hyperparameter search library Optuna.

Next, we will transition from our modern reimplementation of the Random Forest architecture proposed by Hutter et al. to the state-of-the-art machine learning model for tabular data: XGBoost. First, we will report XGBoost's performance on the same data and with the same explanatory variables against our hyperparameter-tuned reimplemented Random Forest to compare the performance achieved *only* by changing the machine learning architecture.

Furthermore, we will modernize the other crucial part of the pipeline, the dataset. We will move from the outdated heterogeneous MIP library provided by Hutter et al. in the pre-2014 era and utilize the current state-of-the-art dataset, MIPLIB, for training and evaluation. The results achieved using this modern library more closely reflect typical MIP instances used today than a library from over a decade ago. This is essential for providing reliable runtime prediction results in ProvideQ. Additionally, we will briefly describe our implementation of survival analysis in XGBoost. This method uses information from censored data because we terminate resource-intensive solvers on MIPLIB instances after one hour to prevent possible infinite calculations due to the NP-hard nature of the problem.

Lastly, we will transition to a description of the code and technical aspects necessary for this experiment. Specifically, we will describe the following implementations in detail:

1. Code for the Python reimplementation of hypertuned Random Forests and XGBoost.

2. Code for runtime measurement with CPLEX 22.1.
3. Changes and modernization of the explanatory-variable extractor by Hutter et al.

This section is divided into the following chapters:

1. 5.1 Migration to Python - Re-implementation of RF proposed by Hutter et al. in modern Python/SciKit-learn and evaluation of Baysian hyper-parameter tuning with Optuna, on old dataset.
2. 5.2 XGBoost on BIGMIX - Evaluation of the newer model, XGBoost, on the same data and reports on the improvements.
3. 5.3 Dataset update to MIPLIB - Switch to modern MIPLIB 2017 data-set, recalculation of runtimes with modern CPLEX & bechmark on XGBoost.
4. 5.4 Survival Analysis - Handling of censored runtimes with XGBoost-AFT.
5. 5.5 Discussion - Summary of findings, and open issues.

We begin with the first part of our contribution: migration from MATLAB to Python.

## 5.1 Migration to Python

Hutter et al.'s experiment was published in 2014; however, the MATLAB experimentation code dates back to mid-2012. By that time, there were already default implementations of many machine learning (ML) algorithms, including Random Forest (RF), in languages such as C and MATLAB. Due to changes to the default RF, as described in RQ1, Hutter et al. implemented their state-of-the-art Random Forest library. Nowadays, however, Python and its scikit-learn library have become the de facto standard for implementing machine learning models, as stated in the NVIDIA Glossary[44]. The advantages that MATLAB offers - namely, that it is strongly typed and compiled, and thus much faster - must be considered. However, we recognize that, because MATLAB is proprietary and not up to industry standards for ML models such as XGBoost, Python seems to be the obvious choice in 2025.

In this section of RQ2, we will first present the results of reimplementing the modified random forest (RF) algorithm in Python using the SciKit-learn framework. The changes made by Hutter et al., particularly those to the node-splitting behavior, are not included in the SciKit-Learn framework. Therefore, we had to write a wrapper to add this custom behavior. We do not expect this step alone to produce better results, since we are only migrating the programming language while leaving the underlying machine learning architecture close to the one written in MATLAB by Hutter et al.

To make a fair comparison with the results presented by Hutter et al., we will evaluate the Python counterpart using the same precomputed explanatory variables and runtime values for the BIGMIX-CPLEX benchmark provided by Hutter et al.

After establishing this baseline, we will adapt the architecture by tuning the hyperparameters of the Python reimplemented model and report on the improvements.

Lastly, we will report on the code and library adaptations needed for the migration and subsequent model improvements. We will describe how we found a suitable SciKit-Learn library counterpart to the RF model implemented by Hutter et al. in MATLAB. We will also explain how we wrote a wrapper to accommodate the custom changes the authors made to the RF architecture that are not available in any SciKit-Learn library. Lastly, we will explain how we wrote the Python code to conform with modern software development standards.

We begin with the results of the migrated model.

**Baseline RF in scikit-learn**

Below in Table 7 we represent the results our modernized Python RF version achieved. We report the performance with 10-fold cross-validation across the error-metrics **RMSE**, Pearson Correlation (**CC**) and Log-likelihood (**LL**) computed on the $log_{10}$ transformed runtimes.

Table 7: Random-forest performance on the CPLEX–BIGMIX scenario: original results of Hutter et al. vs. our Python/scikit-learn re-run.

| Scenario | **Hutter et al. (2014)** | | | **This work** | | |
|---|---|---|---|---|---|---|
| | RMSE | LL | CC | RMSE | LL | CC |
| CPLEX–BIGMIX | 0.64 | −0.63 | 0.90 | 0.658 | −0.94 | 0.905 |

On **RMSE**, we got worse results of about 3.1%. We attribute this lower score to two factors: different seeding libraries, which altered the distribution of the samples used for training and evaluation, and changes to the underlying SciKit-Library code used to create the trees.

On **CC**, we saw statistically insignificant improvements, but on **LL** score, we saw worse results of about 49.1%. We presume possible reasons for this **LL** score include the architectural differences between our scikit-learn library based reimplementation and the original MATLAB version. Particularly, those to node-splitting and variance modifications proposed by Hutter et al. Full details of our implementation are described in Code. LL is sensitive to variance estimates, leading to high jumps in this metric. However, through hyperparameter tuning described in the next section, we managed to turn this around and achieve better results then those reported by the authors.

**Optimization and Hyperparameters**

Hyperparameters define how a specific machine learning (ML) architecture works (compare Figure 2.3). Besides the ML architecture class itself (Trees, Neural Networks, etc.) and the choice of explanatory variables, hyperparameters play the biggest role in the quality and performance of the ML model. In their experiment, Hutter et al. chose the hyperparameters for all the models, including the best-performing regression trees, basing their choices on manual experimentation and domain expertise.

There are many methods to find hyperparameters, some most notable are

- Manual Search: gained through experimentation, possibly not systemized
- Grid Search: systematic tests all hyperparameter combinations in specified range
- Random Search: random search over hyperparameter combinations in specified range
- Baysian Optimization: A probabilistic method which learns sequentially from previous iterations

Although Hutter et al. did not optimize hyperparameters for most of their study, they dedicated a small section to this purpose. Table 8 shows the results of their hyperparameter optimization.

Table 8: RMSE for default ($\lambda_{\text{def}}$) and optimised ($\lambda_{\text{opt}}$) hyper-parameters on the BIGMIX benchmark family. Best value per row in **bold**. RMSE Evaluated on 10-cross-validation with 2-cross-validated hyperparameter-search.

| Scenario | **RMSE** | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | RR | | SP | | NN | | RF | |
| | $\lambda_{\text{def}}$ | $\lambda_{\text{opt}}$ | $\lambda_{\text{def}}$ | $\lambda_{\text{opt}}$ | $\lambda_{\text{def}}$ | $\lambda_{\text{opt}}$ | $\lambda_{\text{def}}$ | $\lambda_{\text{opt}}$ |
| CPLEX–BIGMIX | 3E8 | 0.91 | 0.93 | 0.93 | 1.02 | 0.91 | **0.64** | **0.64** |
| Gurobi–BIGMIX | 1.51 | 1.21 | 1.23 | 1.22 | 1.41 | 1.23 | 1.17 | **1.15** |
| SCIP–BIGMIX | 5E6 | 0.82 | 0.88 | 0.81 | 0.86 | 0.74 | **0.57** | **0.57** |
| lp_solve–BIGMIX | 1.10 | 1.74 | 0.90 | 0.88 | 0.68 | 0.60 | 0.50 | **0.47** |

They used DIRECT [31], an iterative method that divides the search space into smaller rectangles with each step and evaluates the centers of the most promising areas. Looking at the results above, we can see that the hyperparameter search improved their ridge regression model results; however, they did not find any significant improvements for random forest (RF) models. Hutter et al. argue that the time needed for the search is not worth the diminishing returns, especially for RF.

In our experiment, we used the state-of-the-art hyperparameter search framework, OPTUNA (2019) [2]. Several studies, such as [5], show that although RF performs well out of the box, significant performance gains can be achieved through hyperparameter optimization, particularly for gradient boosted models. Furthermore, considering that each CPLEX run takes minutes for each of the 1,050 instances in the dataset, we argue that an investment of, for instance, one hour into a smarter search can quickly amortize itself. In recent years, many Bayesian optimization methods, such as TPE [7] or HyperOpt[6], have been used; however, their interfaces were not user-friendly, and they had limited ability to stop unpromising trials [2]. Optuna changed this with define-by-run logic: Python code that builds hyperparameter search models within Python code without the need for separate schemas, such as JSON. Consider the following example, which specifies the search space for

the number of trees in an interval with a step that defines the difference in value that each trial can make to this parameter. The figure below Figure 20 illustrates the difference in ease of defining the goal in Optuna versus HyperOpt.

```python
import optuna
import ...

def objective(trial):
    n_layers = trial.suggest_int('n_layers', 1, 4)

    layers = []
    for i in range(n_layers):
        layers.append(trial.suggest_int('n_units_l{}'.
            format(i), 1, 128))

    clf = MLPClassifier(tuple(layers))

    mnist = fetch_mldata('MNIST original')
    x_train, x_test, y_train, y_test =
        train_test_split(mnist.data, mnist.target)

    clf.fit(x_train, y_train)

    return 1.0 - clf.score(x_test, y_test)

study = optuna.create_study()
study.optimize(objective, n_trials=100)
```

(a) Optuna

```python
import hyperopt
import ...

space = {
    'n_units_l1': hp.randint('n_units_l1', 128),
    'l2': hp.choice('l2', [{
        'has_l2': True,
        'n_units_l2': hp.randint('n_units_l2', 128),
        'l3': hp.choice('l3', [{
            'has_l3': True,
            'n_units_l3': hp.randint('n_units_l3',
                128),
            'l4': hp.choice('l4', [{
                'has_l4': True,
                'n_units_l4': hp.randint('n_units_l4',
                    128),
            }, {'has_l4': False}]),
        }, {'has_l3': False}]),
    }, {'has_l2': False}]),
}

def objective(space):
    layers = [space['n_units_l1'] + 1]
    for i in range(2, 5):
        space = space['l{}'.format(i)]
        if not space['has_l{}'.format(i)]:
            break
        layers.append(space['n_units_l{}'.format(i)] +
            1)

    clf = MLPClassifier(tuple(layers))

    mnist = fetch_mldata('MNIST original')
    x_train, x_test, y_train, y_test =
        train_test_split(mnist.data, mnist.target)

    clf.fit(x_train, y_train)

    return 1.0 - clf.score(x_test, y_test)

hyperopt.fmin(fn=objective, space=space, max_evals
    =100, algo=hyperopt.tpe.suggest)
```

(b) HyperOpt

Figure 20: Comparison of Optuna vs HyperOpt on the same task [2].

Our RF tuning protocol launched 40 trials, each with a 2-fold cross-validation for hyperparameters and a 10-fold cross-validation test on unseen instances, just like Hutter et al. Our search space included the most commonly tuned parameters: the number of trees, the tree depth, and the criteria for when the nodes stop splitting. We used a fixed seed of ”1234” to ensure reproducible results. Since we were unsure about the size of the search intervals, we first tried the defaults proposed in the literature and then computed a visualization showing which intervals for each parameter yielded the best results across all trials, as depicted below in Figure 21.

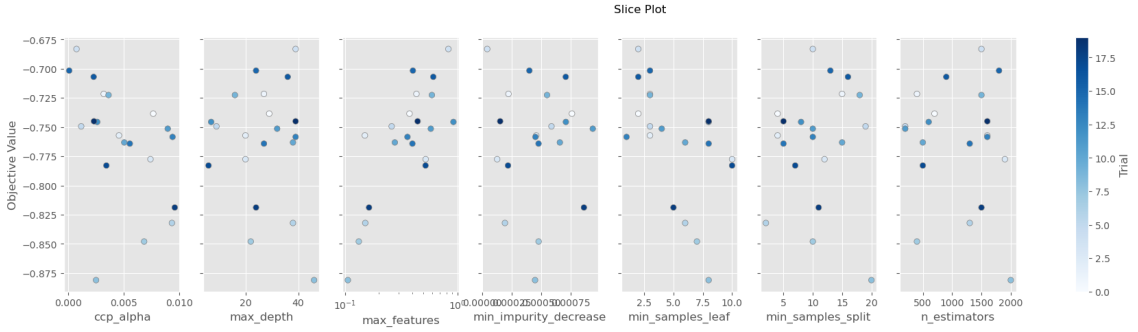



Figure 21: Optuna slice plot for Random Forest Hyperparameter search (40 trials). Each panel shows CV objective vs. a tuned parameter. Points are colored by trial index and highlight the ranges that produced better scores and guided the narrowing of the search space.

Then, by intuition, we shrank the intervals and rerun the experiment to give Optuna a better chance of finding better results.

Optuna delivered the best configuration, which achieved **log-RMSE** improvements of about 4.1%, as seen below:

Table 9: Random-forest RMSE on CPLEX–BIGMIX with default and optimised hyper-parameters $\lambda$. Left: values reported by Hutter et al. (2014). Right: our Python/scikit-learn re-run. Lower is better.

| Scenario | **RF – Hutter et al.** | | **RF – this work** | |
|---|---|---|---|---|
| | $\lambda_{\text{def}}$ | $\lambda_{\text{opt}}$ | $\lambda_{\text{def}}$ | $\lambda_{\text{opt}}$ |
| CPLEX–BIGMIX | 0.64 | 0.64 | 0.658 | 0.614 |

Considering that the hyperparameters for RF are typically effective from the beginning, as demonstrated in Table 8 and the meta-study on ML tree architectures [5], we argue that the 4.1% improvement achieved by the Optuna framework through its automated search will be crucial for the gradient boosted models presented in subsequent chapters.

We can also now compare the results of our optimized hyperparameter search across **RMSE**, **LL**, and **CC**. The results are visible in Table 10:

Table 10: Random-forest performance on the CPLEX–BIGMIX scenario: original results of Hutter et al. vs. our Python/scikit-learn optimized re-run.

| Scenario | **Hutter et al. (2014)** | | | **Optimized RF** | | |
|---|---|---|---|---|---|---|
| | RMSE | LL | CC | RMSE | LL | CC |
| CPLEX–BIGMIX | 0.64 | −0.63 | 0.90 | 0.614 | −0.5773 | 0.9180 |

As we can see, now we achieve improvements of 4.1% on **RMSE**, 8.4% improvements on **LL** and minor improvements of 2% on the **CC** score - beating the MATLAB baseline across all metrics.

**Code**

Below, we will walk through the Python notebook[5] that implements the modernized random forest pipeline, covering everything from data loading to Optuna tuning and visualization.

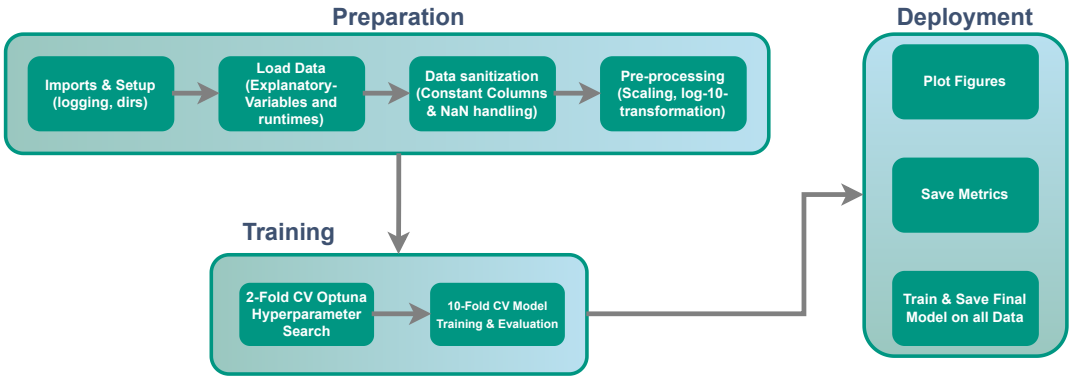



Figure 22: Overview of the random forest modeling pipeline, from data preparation and hyperparameter optimization to model evaluation and deployment.

[5] https://github.com/tubadzin/Performance-Prediction-for-Subroutines-in-Meta-Solver-Strategies

The main strength of Jupyter Notebooks lies in their cells. They allow caching results in each cell and the ability to change other cells without having to reproduce others. This, especially for ML, is useful considering that one may need to load large datasets or perform an extensive hyperparameter search, which would be resource-consuming to do on each program restart. Furthermore, Jupyter Notebooks allow for a lot of experimentation, which is a big part of ML. This differs from Software Development, where experimental artifacts, such as prototypes or vertical slices, are part of a lengthy development process, where more often than not, results can first be seen after hours of active coding. With Python (and its scripting magic, where one can train a trivial ML model in a few minutes with incredibly easy interfaces), and especially with Jupyter Notebooks and cells, it is easy to fall into bad practices. For this, we have persisted each experiment run with crucial information such as code snapshots, models, learned hyperparameters, and visualizations. Furthermore, we tried to adhere to Single Responsibility, where each cell has a clearly defined task.

#### Data

For this experiment we have kept the explanatory-variables and runtime matrices from the original Hutter et al. CSVs.

#### Pre-processing

We keep the pre-processing described in Table 4.3 - z-scoring, and log-transformation of runtimes. We again note that the z-scoring (standardization) should not have any impact on Tree-based ML models, however, to provide clean comparison, we keep this step, as it is part of the pipeline in MATLAB.

#### Model

Scikit-learn provides a library called "RandomForestRegressor", which is the implementation of the default RF algorithm known in the literature [25]. However, we could not use it for the same reasons Hutter et al. changed their architecture (compare Table 4.3):

1. Random splits: In the 2014 paper, every node chooses the best explanatory variable, but then draws the exact split point uniformly at random inside the optimal interval.

2. Log-likelihood: The default RF captures only the mean, while Hutter et al. stored additional leaf-level variance to capture quantitative uncertainty.

We hoped that we could repair (1) with hyperparameters provided in the Scikit-learn RF library; however, as this is a fundamental change to the architecture itself, it is not exposed as a tunable parameter. We found another ML model, called ExtraTreesRegressor [46], which already uses fully random cut points as an architectural decision. Just like RF, ExtraTreesRegressor utilizes an ensemble of decision trees, which use a fraction of explanatory variables at each split. We could not find any default implementation that included the changes described in (2), so we decided to write a wrapper on top of ExtraTreesRegressor.

They add leaf-level caching by walking each tree after fitting, gathering the training targets that fell into each leaf, and storing

$$\mu_l = \text{mean}(y), \quad \sigma_l^2 = \max(\text{var}(y), \sigma_{\min}^2)$$

Setting $\sigma_{\min} = 0.01$ prevents zero variance when a leaf holds just one sample. At prediction time, each tree outputs its leaf mean and variance, giving the same law of total variance described in Table 4.3.

We ensured that our wrapper is fully compatible with all interfaces that interact with scikit-learn's ExtraTreesRegressor by overloading the constructor and parameters. We made changes to the $fit()$ method, which now first trains the underlying ExtraTreesRegressor and then, for each tree, calls $tree.apply(x)$ to get leaf IDs, which are then used to compute the mean and variance. Lastly, we adapted the $predict()$ method so that it collects all $\mu$ and $\sigma^2$, where $\mu$ is the prediction of the ensemble and $\sigma^2$ the additional parameter we utilize to compute the log-likelihood (**LL**), which measures the certainty of predictions.

Our implementation is faithful to the one presented in the 2014 MATLAB code but is now built on open-source Python libraries, which is in fact the default language for ML.

#### Cross-validation

We employ a nested CV, with outer 10-fold CV measuring the final error, where each outer fold creates a two-fold inner CB that Optuna uses for Baysian optimization.

#### Persistance

We wrote *save_model*, *save_optuna* and *save_metrics* helpers. The final model is serialized as JSON, so that it can be easily redeployed in any setting. Each notebook execution starts with a time-stamped directory.

We provide our Jupyter Notebooks code along with further details and documentations at .

## 5.2 XGBoost on BIGMIX - Further Improvements

In this section, we will compare the most promising model, XGBoost, with the results presented by Hutter et al. and the modernized baseline from subsection 5.1 on the same data.

In the previous section, we showcased that a modern RF implementation combined with Optuna managed to improve **log-RMSE** by 4.1% - where we gained the most from the modern state-of-the-art hyperparameter search library Optuna. Based on that, we reason that modernization of the ML model itself can provide further improvements to runtime prediction.

In the discussion of the "Classifying Machine Learning Models" section, we compared various tree-based machine learning (ML) architectures and determined that XGBoost is the most promising model for runtime prediction. XGBoost is used in

many competitions and has demonstrated superior performance in numerous studies. It supports survival analysis objectives. XGBoost uses an iterative process called boosting to grow small trees and identify areas where results are unsatisfactory. Then, it grows the next trees to improve upon the faults of the previous ones. Unlike Gradient Boosted Trees, which use regression trees in an ensemble, XGBoost adds leaf regularization and pruning techniques, as well as a different method for building the trees.

We will now describe the experiment setup and the results XGBoost achieved compared to our modernization and the results presented by Hutter et al.

**Results**

Below Table 11, we first present the results of the default XGBoost implementation without hyperparameter optimization. We report performance using 10-fold cross-validation across the error metrics RMSE and Pearson Correlation (**CC**) on the $log_{10}$-transformed runtimes. We have dropped the Log-Likelihood (**LL**) because the XGBoost library implementation does not store leaf-level variance $\sigma^2$, which was the modification Hutter et al. introduced to their RF and which we adapted in the previous section.

Table 11: Performance on the CPLEX–BIGMIX scenario: original results of Hutter et al. vs. our modern re-runs. No hyperparameter tuning.

| Scenario | **Hutter et al. (2014)** | | **Modern RF (No hyp.)** | | **XGBoost (No hyp.)** | |
|---|---|---|---|---|---|---|
| | RMSE | CC | RMSE | CC | RMSE | CC |
| CPLEX–BIGMIX | 0.64 | 0.90 | 0.658 | 0.905 | 0.594 | 0.923 |

Our non-optimized XGBoost implementation improved **RMSE** by 9.7% and 7.2%, and **CC** by 2.6% and 2%, compared to Hutter et al.'s RF and our modern RF, respectively.

Next, we used OPTUNA to determine the best hyperparameters, following the same methodology described in Optimization & Hyperparameters section. We again compare Hutter et al. against the tuned modern RF and XGBoost.

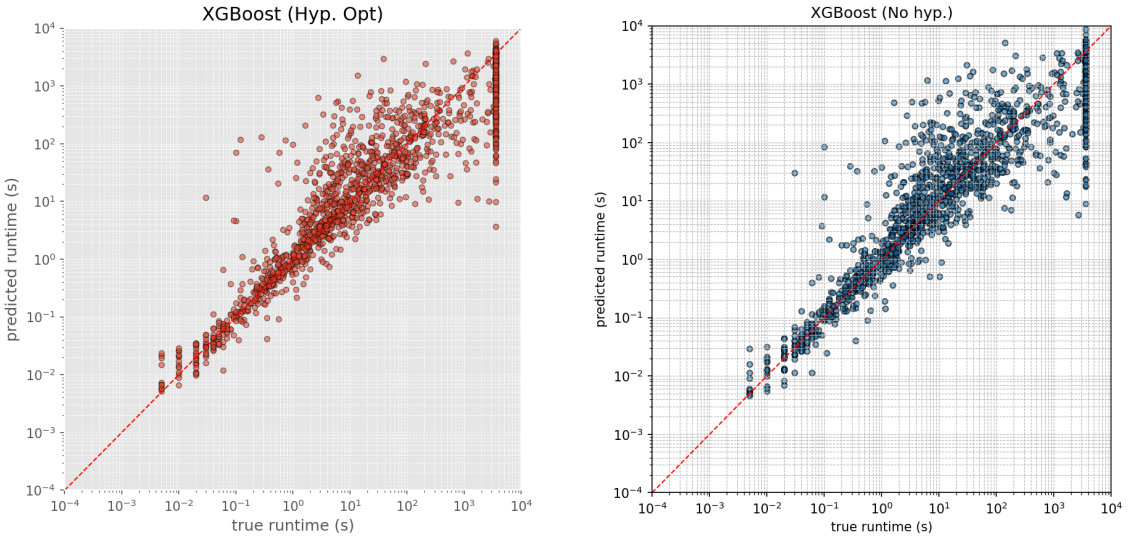
Table 12: Performance on the CPLEX–BIGMIX scenario: original results of Hutter et al. vs. our modern re-runs. Hyperparameter tuning

| Scenario | **Hutter et al. (2014) (Hyp. Opt)** | | **Modern RF (Hyp. Opt)** | | **XGBoost (Hyp. Opt)** | |
|---|---|---|---|---|---|---|
| | RMSE | CC | RMSE | CC | RMSE | CC |
| CPLEX–BIGMIX | 0.64 | 0.90 | 0.614 | 0.918 | 0.588 | 0.925 |

The optimized XGBoost implementation improved **RMSE** by 8.1% and 4.2%, and **CC** by 2.8% and 0.7%, compared to the optimized Hutter et al. and our modern RF, respectively. The most surprising result is the small difference between the default and optimized versions of XGBoost. If we compare the results from Table 11 and Table 12, namely **log-RMSE** values of **0.594** and **0.588**, they do not appear to differ in

a statistically significant way. We expected to see larger improvements, as XGBoost is usually more sensitive to hyperparameter optimization than, for instance, RF [5].

We have also visualized in Figure 23 the results for the 10-fold cross-validated predictions against the true run-times.




(a) XGBoost (Hyp. Opt) (b) XGBoost (No Hyp.)

Figure 23: Visual comparison of XGBoost runtime predictions on BIGMIX.

Both results show that XGBoost can reliably predict algorithm runtime on unseen data. The biggest difficulty can be seen at the censored edges, which is to be expected for the many instances with a censored runtime of 3600 seconds.

**Code**

The Python pipeline with which we compared both models had variation only in the model training and Optuna hyperparameter search parts. In the model training cell, we did not make any significant changes to the interfaces provided by Scikit-learn, as was the case in subsection 5.1.

In the model training cell, we transitioned from using the Scikit-learn wrapper for XGBoost to using the dedicated XGBoost library [66]. The XGBoost API offers more functionality and configuration options than the Scikit-learn wrapper. The original XGBoost library is well maintained due to its wide adaptation, making it reliable to use independently of Scikit-learn while avoiding the minor overhead caused by the wrapper. Lastly, it ensures compatibility with new XGBoost updates.

As RF and XGBoost are two different architectures, they also utilize different hyperparameters. Small differences can be seen in the hyperparameter search space for Optuna, however, only the passed parameters have changed. The rest of the pipeline can be followed in the previous code section.

## 5.3 BIGMIX to MIPLIB2017 - New Benchmark

With the new baseline ML model, optimized XGBoost, we will now move from the BIGMIX explanatory-variable and runtime matrices provided by Hutter et al. and

analyze performance on the current state-of-the-art benchmark dataset: MIPLIB 2017 [21].

We calculate the runtimes of MIPLIB instances using a modern CPLEX version. We feel an update to the runtimes is long overdue, basing our reasoning on the findings from a comprehensive study titled "Progress in Mathematical Programming Solvers from 2001 to 2020" [37]. Koch et al. performed a 20-year retrospective benchmarking study on MIP solvers. They compared the performance of leading solvers starting from the year 2001 and ending in 2020. They showcased a $\times 1000$ speed-up on average over 20 years on MIP instances. The findings show that many instances which were considered unsolvable in 2001 can now be solved in a span of a few seconds.

For explanatory variables, we utilize the modernized explanatory-variable extractor provided by Hutter et al., as their study provided the most extensive research on MIP explanatory variables we could find.

**MIPLIB Dataset**

The most heterogeneous dataset used by Hutter et al., BIGMIX, contained instances collected prior to the year 2012. BIGMIX was composed by Hutter et al. from publicly available MIP benchmarks [27]. However, we could not reproduce it. Instead of the dataset itself, Hutter et al. published only the names of the instances used in the experiment, which also makes it hard to assess its usefulness today.

**Why MIPLIB 2017**

MIPLIB 2017 [21] is the most recent release of the open-source Mixed Integer Programming Library. It was designed and developed by researchers in collaboration with commercial solvers such as CPLEX to create an extensive, balanced benchmark. It contains 1065 (real-world, not synthetic) instances, classified across easy, hard, open, and infeasible categories. MIP covers a wide variety of problem types, such as routing, scheduling, network design, or set covering. For this reason, MIPLIB is composed of instances across many problem domains.

**Runtime Calculation**

We utilized the CPLEX solver, as it is one of the options available in the ProvideQ framework. We plan to extend our ML pipeline with other solvers in the future.

MIPLIB provides, alongside the dataset, metadata such as a list of instances that have not been solved. The number of open instances in the collection is 221 out of 1065. We used this information to exclude these open instances from our runtime calculation script. We treated each open instance as a 3600-second right-censored sample. We retained the easy, hard, and infeasible tiers, a few of which also did not terminate within the 3600-second time limit.

We performed the calculations on CPLEX 22.1, single-threaded, on a MacBook M1 Pro (10-core), macOS 14.4. The calculation took 213 hours to complete for 1065 instances, averaging 268.12 seconds per instance. We chose this hardware setup because it is approximately as fast as the deployment server of ProvideQ, meaning the runtimes displayed by the model on the server are a close approximation.

For the 1065 instances, 405 rows recorded runtimes greater than our cap of 3600 seconds.

**Explanatory-Variable Extraction**

We performed the calculation on CPLEX 22.1 for pre-solve, multi-threaded, on a MacBook M4 Pro (12-core), macOS 14.4. The calculation took 1 hour to complete for 1065 instances, averaging approximately 3 seconds per instance.

We use the modernized version of the explanatory-variable calculator described in Table 2 by Hutter et al. We note that MIPLIB provides an explanatory-variable extractor with 110 variables, which can be categorized into the following groups:

1. Size
2. Variable types
3. Objective function
4. Variable bounds
5. Matrix non-zeros
6. Matrix coefficients
7. Row dynamism
8. Sides (RHS/LHS)
9. Constraint classification
10. Decomposition

We observe a high overlap with Table 2; however, we notably see a lack of probing and timing explanatory variables, which showed a performance gain of about 26% in the Hutter et al. [27] study. We see potential in combining the explanatory variables introduced by MIPLIB and those by Hutter et al. - we leave this, however, as future work.

**Results**

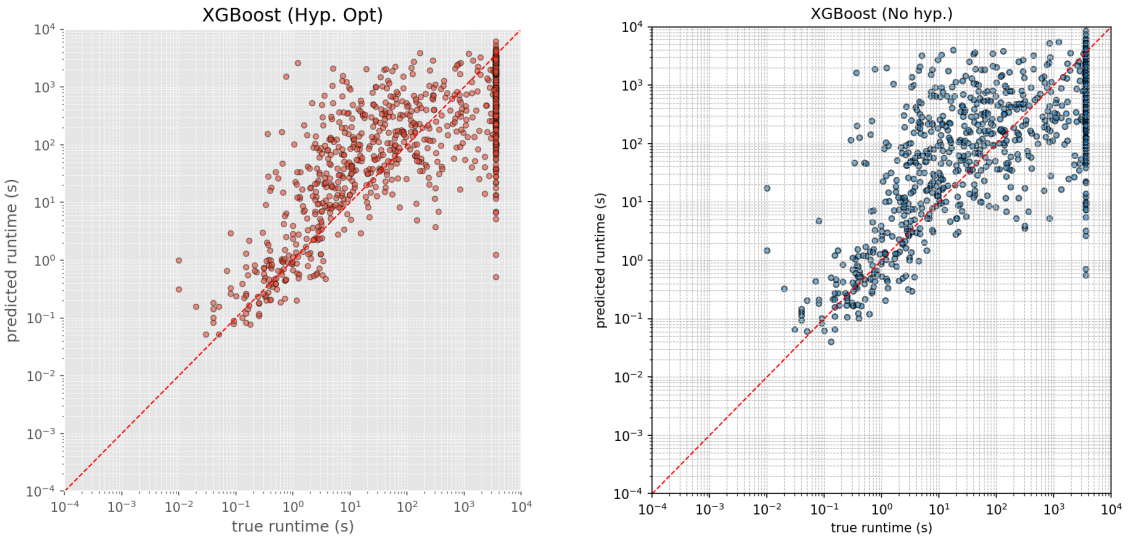
Below, in Table 13, we present the results for both the default and hyperparameter-optimized XGBoost implementation on the MIPLIB 2017 dataset. We report performance using 10-fold cross-validation across the error metrics **RMSE** and Pearson Correlation (**CC**) on the $log_{10}$-transformed run-times.

Table 13: Performance on the MIPLIB 2017 dataset: XGBoost with and without hyper-parameter optimization.

| Scenario | **XGBoost (No Hyp.)** | | **XGBoost (Hyp. Opt.)** | |
|---|---|---|---|---|
| | RMSE | CC | RMSE | CC |
| MIPLIB 2017 | 1.005 | 0.740 | 0.9341 | 0.7780 |

Hyperparameter tuning provided results more in line with our expectations from [5] - **log-RMSE** improves by about 7%, and **CC** is higher by 5%. We can now see the influence of the dataset on predictive performance. A **log-RMSE** of $\approx 1$ means the true predictive error is on average $\times 8.6$ the actual recorded run-times. We argue that this score is much more representative of the true difficulty of MIP run-time prediction than the multiplicative error of $\times 4$ achieved on BIGMIX. It confirms our suspicion that MIPLIB is a more heterogeneous and thus harder-to-predict dataset.

We now examine the visualized results in Figure 24 for the 10-fold cross-validated predictions against the true run-times:




(a) XGBoost (Hyp. Opt) (b) XGBoost (No Hyp.)

Figure 24: Visual comparison of XGBoost runtime predictions on MIPLIB.

Looking at the hyperparameter-optimized figure (a), we see a high number of points to the left of the vertical $\mathbf{10^2}$ grid-line that are pessimistic relative to the perfect prediction **45°** reference, with most points falling within the pessimistic **×8.6** under-confident prediction. However, if we consider the points to the right of the vertical $\mathbf{10^2}$ grid-line, we can observe how the trend changes. Now, the model begins to predict optimistic run-times, as most predicted run-times lie below the perfect prediction 45 reference.

If we compare Figure 24a with Figure 23a, we see that on BIGMIX, XGBoost was able to balance the pessimistic and optimistic predictions, whereas on MIPLIB it is more unbalanced, reflecting the much lower **CC** score.

Lastly, points beyond the **3600** seconds mark show the same trend across all datasets and ML models. All runs that exceed the **3600** seconds mark are censored with exactly this value. Therefore, the model sees a wall at this cut-off and has no gradient to learn better predictions.

If we were to deploy this model, we suggest the following rule of thumb: considering that the true predictive error is on average **×8.6** the actual recorded run-times, we would treat every prediction **> 350 seconds** as a warning: likely long or unsolved. The reasoning is that with a **×8.6** error, a **350**-second estimate could still mask a **3600**-second censored run. However, all predictions with shorter run-times can be considered relatively comfortable within the margin of predictive error.

Now with established performance on MIPLIB, we explain and describe the implementation that produced the explanatory-variables and run-times used in the experiment above.

### Code

This section documents and describes the implementation used to generate MIPLIB runtimes and calculate explanatory variables. We supplement our documentation with figures.

**Runtime-Calculator**

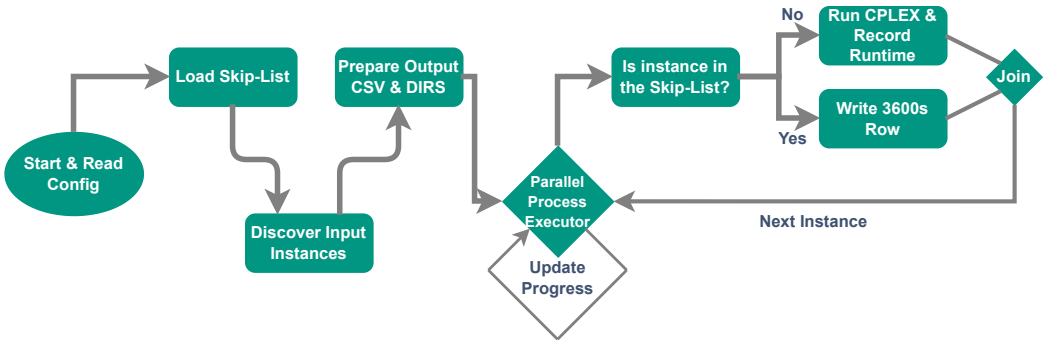



Figure 25: CPLEX Runtime Calculator Code Flow

We have written a Python script that measures CPLEX runtimes in a single-threaded environment. The script takes a list of open instances to preload these instances and writes the results directly to a CSV with a fixed runtime of 3600 seconds.

Each CPLEX run is invoked with a 3600-second time limit, with a sub-process controlling the CPLEX execution cap at 3610 seconds to account for I/O lag. We run each instance one at a time to avoid cache issues and to allow CPLEX to utilize the full compute power—high parallelism would distort run-times.

We have added the tqdm library, which shows the current progress and total calculation time for all instances.

The CSV output contains two rows, one with the instance name and the other with the calculated run-times. We made the script atomic, meaning each header row is written once, and the results are flushed instantly. This allows the script to be rerun after a failure while still retaining partial results. This was crucial given the size of the dataset we processed, where one script invocation could take up to several days.

**Explanatory-Variable Extractor**

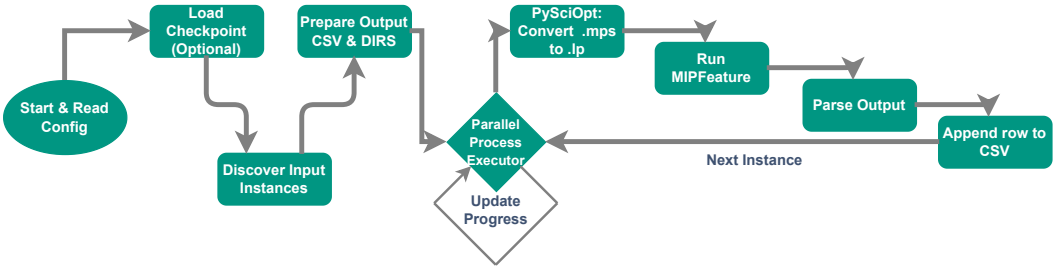



Figure 26: MIPfeature Extractor Code Flow

We have wrapped the modernized MIPfeature binary in a modern Python script. Each instance in the MIPLIB collection is wrapped in a $*.mps.gz$ file. For each instance, to avoid memory blow-out, we unzip dynamically only when the instance is called and we keep the output cached in script memory for faster processing.

Since we utilize the explanatory-variable extractor provided by Hutter et al., we had to transform the instances from *.mps* to *.lp* format - for this, we used the open-source SCIP Python API.

As explanatory-variable extraction is not as resource-intensive as runtime calculation, we composed the script to automatically detect the number of threads on the machine and utilize all but one, for fast performance. Each instance is capped at 60 seconds, which shouldn't be exceeded under normal circumstances.

As in the Runtime-Calculator Wrapper, we implemented atomic CSV writing, in case the explanatory-variable extractor must be run on large datasets.

## 5.4 Right-Censored Data and Survival-Analysis

Predicting the runtime of solvers is challenging, especially when the target variables (solver run times for a given instance) used for training are limited by a time cap (3600 seconds) to save computation time and gain a large training dataset. The time measured for instances that did not terminate within the time cap is called a censored runtime because we only know the lower bound but not the exact time that the solver would have taken. One approach is to record the capped runtime as the actual runtime, which we have used in this thesis thus far. Another approach is to discard this data; however, this would cause the model to treat instances that take a considerable amount of time as if they were faster than they are [27]. Another variant is Survival Analysis, which explicitly handles censored information.

Candidates for survival analysis modeling include Kaplan-Meier (KM), Cox regression (CR), and accelerated failure time (AFT) models.

In this section, we will analyze KM, CR, and AFT methodologies to determine their applicability to our problem of creating empirical performance models for runtime prediction on the MIPLIB dataset using XGBoost. Next, we will focus on the most promising methodology, AFT. We will describe the adaptations needed to make it work with XGBoost, as well as introduce new evaluation metrics. This shift from treating 3600-second runtimes as "true runtimes" to considering them in terms of lower and upper bounds poses difficulties for evaluation using the metrics introduced so far in the thesis. Where possible, we will draw comparisons between the AFT-XGBoost and the squared-error loss introduced by XGBoost in the previous section, explaining why the new variant is potentially better for our use case of runtime predictions for the NP-hard MIP problem.

We will now briefly compare each candidate for our use case of runtime prediction with XGBoost on the MIPLIB data.

### Kaplan Meier

Using the Kaplan-Meier method, we can model the probability that an instance will terminate the calculation after x seconds. We will recall the criteria for a valid prediction made in subsection 3.5. In solver contexts, censoring is dependent because if the solver terminates early due to a time limit, it correlates with difficulty. Furthermore, censoring is informative because we set a global limit of 3600 seconds for our solver. Additionally, we could not model KM with explanatory variables because the model requires only one explanatory variable for each graph. We could

aggregate the results for all explanatory variables, but this would not capture the complexity of MIP instances. Lastly, KM cannot make assumptions beyond censored data, which would not be useful for predicting the runtime of difficult instances with censored data.

#### Cox-Regression

Cox regression is used to model the effect that multiple explanatory variables have on runtime, where some of the target variables can be censored. A major advantage of CR over KM is the ability to model multiple explanatory variables simultaneously. However, CR is sensitive to highly correlated variables and can only model log-linear correlations between different explanatory variables. The variables we use for MIP, however, are more tightly correlated. Lastly, even though the results are interpretable, they compare the effects that different explanatory variables would have on run-times by calculating hazard rates. While it is possible to extrapolate from this the predicted run-times, this method is not very reliable, as noted in many empirical studies, such as that of Hutter et al. [27].

#### Accelerated Failure Time Model

We use the accelerated failure time (AFT) model because it directly predicts solver runtimes rather than survival curves, as is the case with the Kaplan-Meier model or relative runtime rates predicted by the Cox regression model. AFT naturally handles censored runtimes by treating them as lower bounds. It also predicts per-instance medians and prediction intervals in seconds. We focus on the AFT objective in the XGBoost library because it provides these predictions. Furthermore, the AFT objective considers all explanatory variables, resulting in the most accurate runtime predictions.

#### Experimental Setup

The setup matches the previous experiment of XGBoost on BIGMIX, with small differences. We use 10-fold cross-validation, where inside of each loop we run Optuna to tune hyper-parameters and we reserve a further 10% subset of the training split. The main changes lie in the regression set-up.

Previously, we trained XGBoost with squared-error loss, in which the model learns the true target values by minimizing the squared error between the predictions and the run-times. However, this method only works when the target is a true value. On MIPLIB, we set the solver runtime to 3600 seconds. This means that run-times with this value do not represent the actual runtime, but rather a lower bound of $< 3600$. Treating 3600 seconds as the true value introduces a bias that causes the predictive performance to favor lower run-times. We implemented the survival AFT objective, in which the model fits a distribution for log-transformed run-times. This allows each training point to be represented as an interval:

$$[y_{lower-bound}, y_{upper-bound}]$$

This yields the following for instances that are solved within the time limit:

$$y_{lower-bound} = y_{upper-bound} = log(\text{true runtime})$$

For censored instances, we have:

$$y_{lower-bound} = log\ 3600; \qquad y_{upper-bound} = \infty$$

The training maximizes the log-likelihood of these intervals so that the censored data are not handled as if they finished at a runtime of 3600 seconds. The final predictions are given as the median of the fitted distribution in seconds.

For evaluation, we moved from **log-RMSE** for the same reasons mentioned above; **log-RMSE** is only well-defined on solved instances. We use AFT-negative log-likelihood (**NLL**), which is calculated using the predicted runtime distribution against the predicted intervals of each instance.

For comparison with earlier sections, we also report **RMSE** and **CC** scores, but *only* on instances for which the true runtime is known, i.e., uncensored run times.

We also introduce additional metrics, which are listed below:

1. Timeout Detection as **Precision**= $\frac{TP}{TP+FP} \in [0, 1]$; higher is better.
2. Sensitivity on censored run-times as **Recall**= $\frac{TP}{TP+FN} \in [0, 1]$; higher is better
3. Concordance index as **C-index**$\in [0, 1]$; higher is better.

**Precision** (1) answers: Of the instances the model flagged as timeouts, what fraction were actually timeouts? Formally:

$$\text{TP} = |\{i : y_i \geq 3600,\ \hat{y}_i \geq 3600\}|$$

$$\text{FP} = |\{i : y_i < 3600,\ \hat{y}_i \geq 3600\}|$$

**Recall** (2) answers: of all the real timeouts, what fraction did the model successfully flag? Formally:

$$\text{FN} = |\{i : y_i \geq 3600,\ \hat{y}_i < 3600\}|$$

**C-index** (3) compares pairs of instances to determine whether the model correctly predicted the faster instance to be faster than the slower ones. Unlike Pearson's correlation coefficient, which we introduced in RQ1 evaluation metics, it can handle right-censored data by only comparing compatible intervals.

### Results

The Table 14 below presents the results that ATF-XGBoost achieved on the MIPLIB dataset.

Table 14: Performance on the CPLEX–MIPLIB scenario (AFT–XGBoost).

| Scenario | **AFT–XGBoost** | | | | |
|---|---|---|---|---|---|
| | NLL | C-index | Precision | Recall | UnderPred |
| CPLEX–MIPLIB | 5.66 | 0.8 | 0.768 | 0.321 | 0.679 |

The model achieved a **negative log-likelihood (NLL) of 5.68**, where a lower score is better. Currently, we lack a baseline against which to compare this score. However, we will use this value as a reference point for our future analyses of scores between subroutines, as well as a possible reference for improvements to the AFT-XGBoost itself in future work.

The **C-index of 0.8** represents a strong correlation; the model correctly orders 80% of the comparable pairs. This indicates that the relative comparison of run-times between instances can be trusted, even if the exact run-times are difficult to predict.

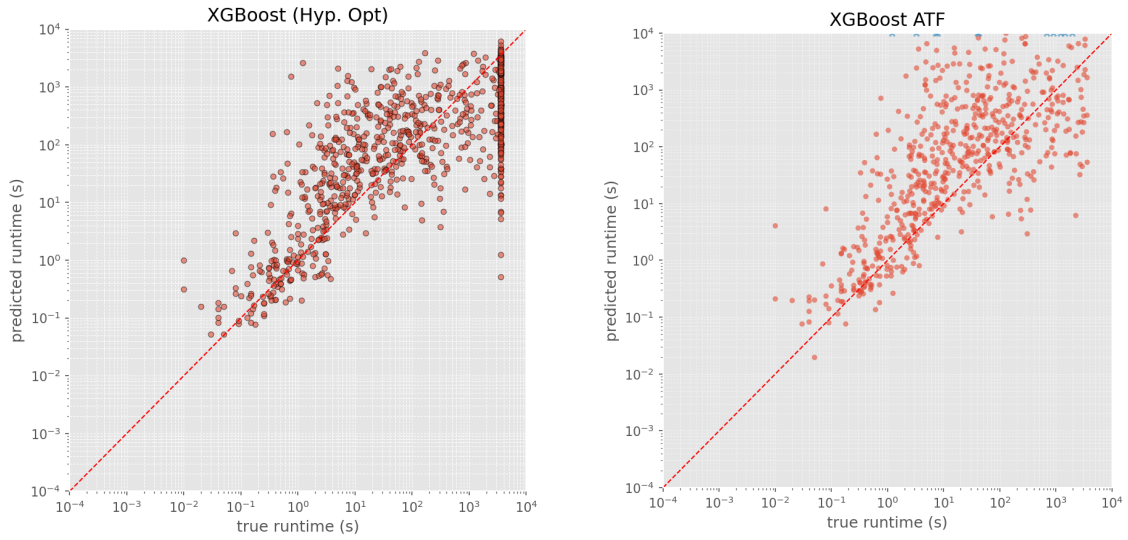
The reported **Precision of 0.752** can be interpreted as follows: out of the instances the model $flagged$ as timeouts, about $\approx \frac{3}{4}$ corresponded to truth.

However, the low **Recall of 0.329** indicates that the model correctly identifies true timeout instances only in approximately $\approx \frac{1}{3}$, meaning it still has a a slightly optimistic view of harder instances.

In the Figure 27 below we plot predicted versus true runtime on MIPLIB for two models: (a) the XGBoost regressor from Section 5.3 trained with squared error and (b) XGBoost with the AFT Survival Analysis objective. For (b) we do not plot the predictions for censored cases because we treat their run-times as unknown in range

$$[y_{lower-bound}, y_{upper-bound}] = [3600\ seconds, \infty]$$

which cannot be interpreted as a ”true runtime”.




(a) XGBoost (b) XGBoost AFT

Figure 27: The red line represents the perfect prediction. Points beyond the plotting window are marked with a ×.

Due to not plotted censored run-times, we can not directly compare AFT model (b) with (a). However, we note one crucial detail: the number of run-times denoted with × which fall beyond the plotting window in the (b) variant. This means that this new variant of XGBoost with AFT tends to behave more conservatively by proposing high run-times further right in the graph in more cases, as opposed to the previous implementation (a).

This tendency of the AFT Model for conservative predictions is even more evident in the Table 15 below:

Table 15: Performance on the CPLEX–MIPLIB scenario: AFT–XGBoost vs. XGBoost.

| Scenario | **AFT–XGBoost** | | | | **XGBoost** | | | |
|---|---|---|---|---|---|---|---|---|
| | RMSE | CC | Precision | Recall | RMSE | CC | Precision | Recall |
| CPLEX–MIPLIB | 1.0658 | 0.6956 | 0.786 | 0.321 | 0.9341 | 0.7780 | 0.900 | 0.077 |

AFT-XGBoost raises **Recall from** 7.7 to 32.1% while reducing **Precision from** 90 to 78.6%. We argue this is a fair trade-off, because catching more true timeouts is worth a increase in false timeout reports when dealing with NP-hard MIP instances.

Lastly, we compare the both models on **RMSE** and **CC** in the following Table. An important aspect is the compatibility of the results. For the AFT variant, we only calculated the **RMSE** and **CC** on the solved (uncensored) instances because we cannot interpret the run times of censored instances as ”true run times”.

Table 16: Performance on the CPLEX–MIPLIB scenario: AFT–XGBoost vs. XGBoost.

| Scenario | **AFT–XGBoost** | | **XGBoost** | |
|---|---|---|---|---|
| | RMSE | CC | RMSE | CC |
| CPLEX–MIPLIB | 1.0658 | 0.6956 | 0.9341 | 0.7780 |

The **RMSE** and **CC** are better for the previous model. Treating timeouts as exactly 3600 seconds provides the previous model with an easier target value for difficult instances, in which predicting $\approx$ 3600 seconds improves the **RMSE** and **CC**. Furthermore, as seen in Figure 27b, the AFT model is more conservative for longer run times, which lowers the metrics further.

Overall, since AFT learned from censored data, its predictions are more conservative near the 3600-second cap. We argue that AFT maintains good prediction accuracy while producing better results for difficult instances. This is crucial, considering the NP-hard nature of the MIP, which can result in potentially infinite solver run times.

#### Technical

To make AFT work, we had to adapt our architecture presented in Code for XGBoost on MIPLIB. We list the most notable changed below.

1. Problem Domain: We transitioned from treating runtime prediction as a standard regression on $log_{10}(runtimes)$ with a squared error objective to a censored survival problem using an AFT objective on *seconds*, with lower and upper bounds to encode timeouts. Training on seconds is justified because the AFT model internally transforms the target variables with the log function.

2. Tuning: In each cross-validation fold, we reserved an additional 10% of the data to enable early stopping and live learning curves, which we will present below.

3. Evaluation: Besides implementing new performance metrics, we also implemented an aggregated plot of learning curves to track whether our model reaches its maximum capacity due to the lack of a comparison against other models.

In the results section, we noted that, at this time, we lack a reference against which to compare the new negative-log-likelihood (NLL) metric. To address this, we have implemented new plotting, as mentioned in points 2 and 3.

These plots show the aggregated rate of improvement curves over 10-folds. To achieve optimal results, we implemented an early stopping strategy. Early stopping stops the creation of further boosted trees after the score measured on the reserved 10% validation data stops improving for $K$ rounds. XGBoost then retains the best iteration (i.e., the best number of trees) found thus far. This prevents overfitting on the training data and saves time due to possible diminishing returns after $K$ rounds. The results are presented in the Figure 28 below:

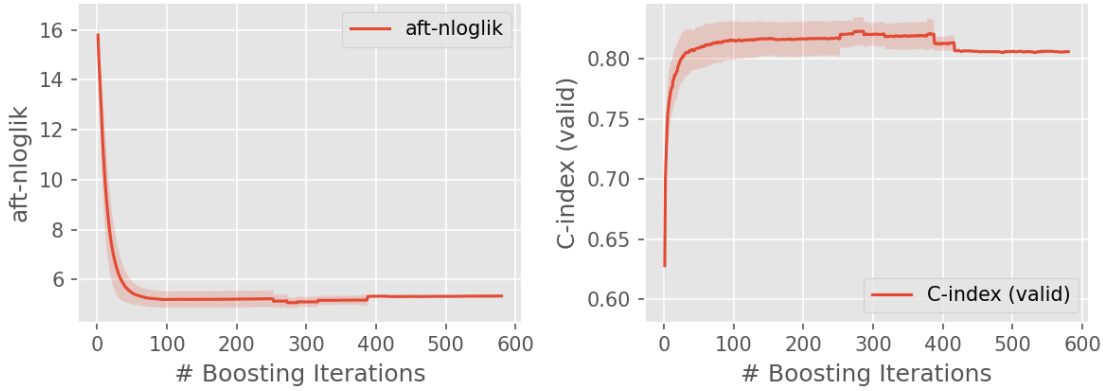



Figure 28: Learning early stopping curves for AFT-XGBoost on MIPLIB. Left aft-nloglik (lower is better). Right validation C-index (higher is better).

The left figure shows the error objective dropping quickly and then flattening for the negative log likelihood. This assures us that the model has reached its maximum performance with our optimized hyper-parameters. The right panel shows the C-index, which exhibits the same behavior. Overall, we conclude that there are no obvious improvements to be made, for instance with longer training, to our current implementation. However, we note that a more detailed analysis of survival analysis is needed, which we will address in future work.

## 5.5 Discussion

**Evaluation insights for ProvideQ**

We see that the jump from BIGMIX to MIPLIB considerably decreased accuracy. A multiplicative error of $\approx 8.6$, which is higher than the $\times 3.8$ we published on BIGMIX,

is arguably a truer reflection of how modern Empirical Performance Models for MIP run-time predictions actually perform.

For practical deployment, we propose a simple heuristic. Because the predictions drift by $\approx 8.6$, which is nearly one order of magnitude, all predictions above 418 seconds should be flagged with a warning: "high-risk: may run for an hour or fail to terminate." A prediction under that 418-second threshold, when multiplied by the average multiplicative error of $\times 8.6$, on average fits the time window of $[40, 3595]$ seconds, which is below the 3600-second cut-off.

Furthermore, we discovered that using censored information improves predictions of worst-case scenarios for data near or at the 3600-second solver timeout cap. On MIPLIB, AFT-XGBoost increases **Recall** for timeouts from 7.7 to 32.1% while **Precision** drops from 90 to 78.6%. We note that, for ProvideQ users, missing a true timeout is more costly than a false alarm. For this reason, we recommend implementing AFT-XGBoost as the model.

### Open Issues

#### Solver Variability

We note the problems outlined in the 2015 publication by Barry Hurley and Barry O'Sullivan [26]. The authors analyzed SAT solvers using SAT competition data to determine the reliability of the results. They discovered that even the best SAT solvers have heavy-tailed run-time distributions. We can explore this in Figure 29 below:

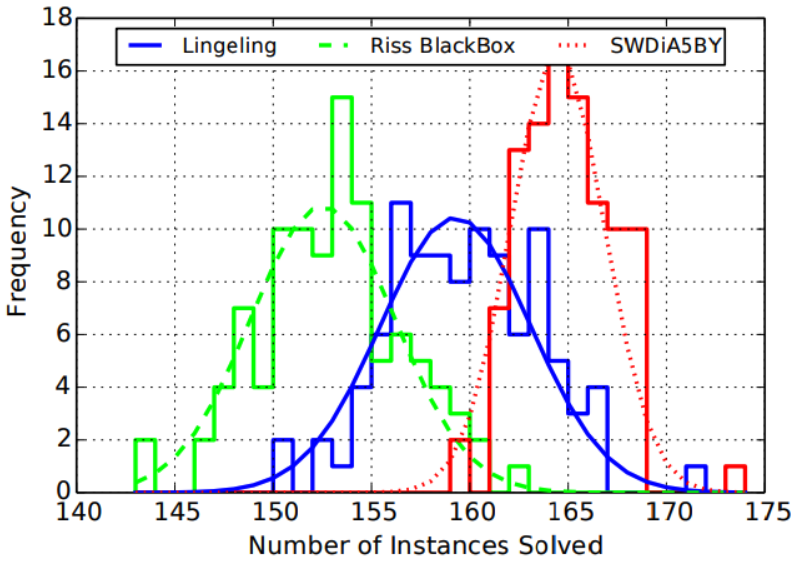



Figure 29: Histogram of the number of instances solved across 100 runs on 300 instances from the application category of the SAT Competition 2014. The three solvers are the top-ranked solvers from the competition [26].

The histogram above shows that any of the solvers placed in the top 3 could have won the competition. Given enough repetition, running the same SAT solver on the same instance can yield dramatically different run-times (extrapolated from solved & unsolved criteria). This variance is noted to stem from the randomized heuristics solvers utilize in each run.

We recognize that this variability is a problem for reliable runtime prediction. We propose, as future work, to improve our ML model by collecting data for each instance multiple times and then incorporating the gathered information into the training pipeline to obtain more reliable results.

### MIP Heterogeneity

MIP is an umbrella term for problems that can stem from domains such as routing, scheduling, network design, or set covering. Throughout the study by Hutter et al. [27], they reported ever-decreasing performance measured in **RMSE** when moving from homogeneous datasets like $CORLAT$ to more heterogeneous datasets like $BIGMIX$. We confirmed this correlation in our study when analyzing performance on $MIPLIB$, the most heterogeneous dataset we have found.

We recognize that further improvements can be gained through a larger dataset, inclusion of variance as mentioned in the paragraph above, better explanatory variables, or ML model fine-tuning. However, in our future work, we wish to propose a different methodology - instead of chasing diminishing returns through the aforementioned improvements, we could leverage the heterogeneity that datasets provide. In the ProvideQ Framework, most users are roughly aware of what class of problems their MIP encodes. We would like to explore the idea of adding a selection tab for each of the solvers, where users can choose the ML model they prefer, as depicted below:

1. Generalized-Model
2. Routing-Model
3. Scheduling-Model
4. Network-design-Model
5. Set-covering-Model

This approach would balance generalization, freedom, and ease of use for novice users while also providing better performance for domain experts.

# 6 Machine Learning Based Performance Prediction for Hybrid Solvers (RQ3)

As the popularity of hybrid quantum-classical algorithm frameworks grows, the field of quantum algorithm performance prediction is also emerging. In previous RQs, we analyzed the potential of EPMs for runtime prediction on classical algorithms, specifically MIP. Our results provided insight into the best-performing model and the importance of explanatory variables. However, quantum optimization problems differ from their classical counterparts in many ways. In this section, we will analyze how much of the knowledge gained in previous RQs, which addressed runtime prediction ML models for MIP, can be applied to the quantum domain. To do so, we will examine two promising publications by Moussa et al [40] and Volpe et al. [61], which address ML in the quantum domain.

Until now, we have focused only on runtime prediction because the goal of ProvideQ is to empower users to make informed decisions about which solvers to use. Runtime prediction has been the subject of many publications due to its importance when considering NP-hard algorithms that may never terminate. However, many publications on empirical performance models concentrate on supervised machine learning (ML) classification as opposed to regression (as is the case with runtime prediction) because the black-box character that classification proposes is sufficient in most settings where the goal is to achieve the best possible results and the exact metrics that influenced the decision do not need to be disclosed.

To empower users with information and balance the need for additional prediction metrics, such as solver quality, which is essential when approaching the domain of approximation optimization algorithms in quantum-classical contexts, we will expand our run-time prediction criteria by exploring quality prediction. We will extrapolate how to best obtain good runtime and solution quality predictions from the results proposed by Moussa et al. and Volpe et al. for the best model classifiers in quantum-classical machine learning.

Lastly, we will outline our future work on implementing ML for hybrid quantum-classical algorithms in ProvideQ.

## 6.1 Runtime-prediction for quantum sub-routines

We will analyze whether the empirical performance models that worked for classical MIP solvers (which are similar to the explanatory variables for VRP, TSP, Knapsack, Max-Cut, and SAT problems, all of which are provided by ProvideQ) can be transferred to quantum routines, such as those that utilize QUBO as an input format. If not, we will determine what needs to change.

The main driving question is whether runtime prediction is plausible for quantum subroutines. Current NISQ hardware operates QAOA and VQE algorithms, which are designed to address quantum noise. These algorithms are suitable for small-scale problems. However, larger instances are infeasible to calculate on quantum simulations and true quantum hardware due to slow performance. Combined with the fact that translating classical formats into QUBO carries significant overhead, this means that, in the short term, almost every problem instance would be better solved with classical algorithms in terms of runtime. One possible use case for

quantum runtime prediction is within quantum subroutines themselves, as illustrated below in Figure 30 between Qiskit and the D-Wave annealer.

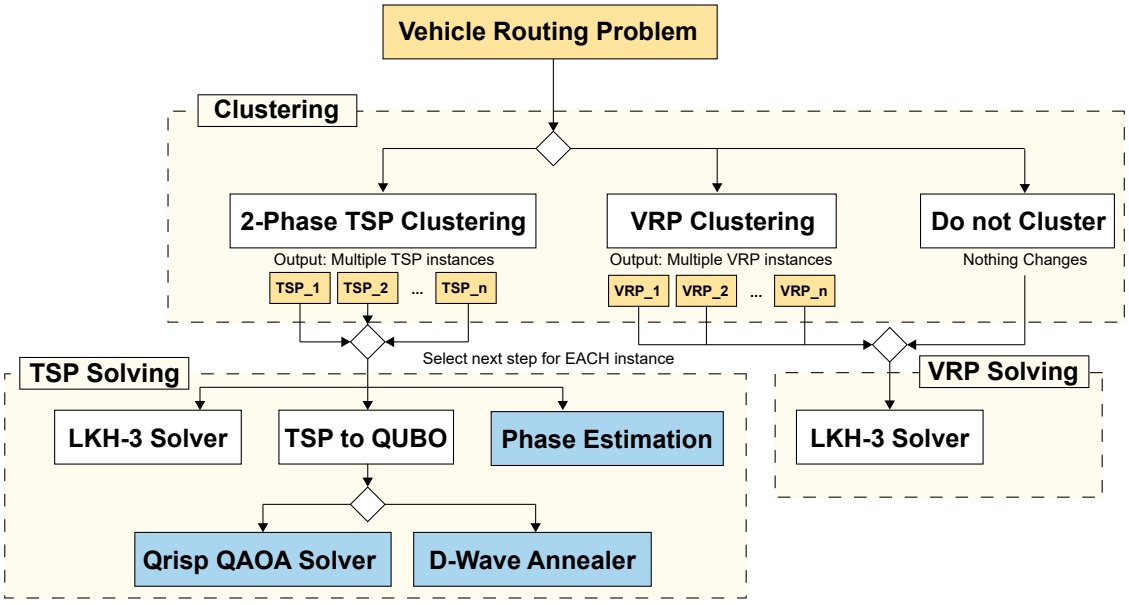



Figure 30: Running Example for a Vehicle Routing Meta-Solver Strategy [16]

Subsequently, we will analyze the viability of quantum-classical runtime performance and compare it to algorithm selection classifiers and current publications in the quantum-classical machine learning (ML) space.

## 6.2 To Quantum or Not to Quantum

Moussa et al. (2020) published "To Quantum or Not to Quantum" (2020) [40], in which they examined an ML classifier learned by an automatic process that selects the best solver/algorithm: The classifier uses either the Quantum Approximate Optimization Algorithm (QAOA) [18] or the Goemans-Williamson (GW) algorithm [22] for a given Max-Cut instance. The classifier uses graph-spectral and GW performance explanatory variables. Their approach results in $\approx 98\%$ accuracy.

In this section, we will evaluate their approach to automatically training ML models and analyze the use of explanatory variables gained from the original, non-transformed instances in ProvideQ.

### Motivation

The authors explain that, in the current NISQ era, the limitations of the hardware include qubit count, gate fidelity, connectivity, noise, and coherence time. However, approximate algorithm families, such as QAOA, can overcome these limitations.

Regarding MaxCut, the authors state that, when defined as an approximation algorithm, QAOA yields worst-case solutions with about 0.69% accuracy, whereas the best classical approximation, Goemans-Williamson, yields solutions with about 0.878% accuracy. Furthermore, it has recently been formally proven that the Goemans-Williamson algorithm outperforms the QAOA algorithm for any given Max-Cut problem.

The authors acknowledge, however, that the formal proof is theoretical, meaning there are instances in which QAOA can outperform GW. For this reason, the authors do not view QAOA as an approximation algorithm but rather as a heuristic optimizer that, when applied to the right instance, can outperform GW.

The authors have formulated the following two research questions:

1. Does QAOA beat the GW algorithm on a certain instance at all?
2. Does QAOA do very well on this instance outperforming GW by a high margin?

Considering that QAOA is expensive to formulate and run on quantum hardware due to classical parameter tuning and quantum circuit evaluation, answering these two questions is crucial. The first RQ establishes whether the quantum QAOA is worth trying, and the second RQ identifies rare instances in which QAOA is not only better than GW but also correct to within 2% of the optimum and outperforms GW by at least 2%.

To answer these questions, the authors used machine learning (ML), specifically an ML algorithm selection model defined as a classifier. Given an instance, this model predicts whether QAOA will outperform GW in terms of solution accuracy on MaxCut.

**Explanatory-Variables**

To make these predictions, the authors engineered 20 explanatory variables that capture various properties of Max-Cut instances. These variables can be grouped into three categories: (i) graph spectral properties, (ii) subgraph-based metrics, and (iii) performance metrics of the GW algorithm. Note that some features in category (ii) are NP-hard to calculate, and these explanatory variables are calculated using the original Max-Cut problem instance rather than the QUBO reformulation.

**Models**

The authors experimented with several traditional machine learning (ML) models, including tree-based gradient boosting, random forest, light gbm, and xgboost [11, 20, 25, 33]. However, they noted that these models achieved an accuracy of less than 63 percent on RQ1, possibly due to the small dataset. The final pipeline used by the authors is an AutoML search with TPOT.

TPOT [45] is a framework that automatically designs end-to-end Scikit-learn pipelines. Essentially, it considers the selection of data preprocessing steps, explanatory variable selectors, machine learning families (e.g., trees, neural networks), and hyperparameter searches as a unified optimization problem. TPOT uses genetic programming to evolve pipelines that maximize a user-provided metric, such as RMSE, against the hold-out data. TPOT utilizes both simple, low-variance learners and more expressive models. Its strength lies in its ability to discover strong machine learning (ML) solutions on small or noisy datasets.

The resulting TPOT code for the first research question is listed below in Listing 1.

```
exported_pipeline = make_pipeline(
    make_union(
        Normalizer(norm="l2"),
        FunctionTransformer(copy)
    ),
    StackingEstimator(
        estimator=KNeighborsClassifier(
            n_neighbors=41,
            p=1,
            weights="uniform"
        )
    ),
    MultinomialNB(alpha=0.1, fit_prior=False)
)
```

Listing 1: Exported pipeline (Criterion 1). Adapted from Moussa et al. [40].

As we can see here, the automatic pipeline is for supervised classification (MultinomialNB predicts discrete classes) and involves two machine learning methods. K-nearest neighbors (K-NN) uses labeled data to predict new, unseen data by grouping the $k$ nearest samples and assigning a label through the majority vote of those labels for classification. Multinomial Naïve Bayes models each explanatory variable independently, meaning the value of one explanatory variable does not influence the occurrence of the others. It then creates a multinomial distribution learned by maximum likelihood estimation, which calculates the probability that a sample in the dataset will be classified as a certain class. The above-depicted pipeline utilized stacking, in which the k-NN model's predictions were used as input for the Multinomial Naïve Bayes model.

#### Results

This machine learning pipeline, which was learned automatically, achieved 4-fold cross-validated results of approximately $\approx 96\%$ for RQ1 and $\approx 83\%$ for RQ2.

They also experimented with the importance of explanatory variables. For the first criterion, meaning whether QAOA can outperform GW on a given instance, they learned that two inexpensive GW algorithm performance metrics are sufficient to predict with $\approx 96\%$ accuracy whether QAOA will outperform GW. Adding all NP-hard graph features provided no additional benefit to this prediction. For the second criterion, meaning whether QAOA can achieve very high accuracy on a given instance while also outperforming GW by a considerable margin, they learned that only graph spectral properties are necessary for the highest score, while subgraph explanatory variables and GW performance metrics had no influence on predictive performance.

## 6.3 QUBO Solver Selector

In "A Predictive Approach for Selecting the Best Quantum Solver for an Optimization Problem" [61], Volpe et al. automated the quantum-classical solver selection

using supervised machine learning (ML). Their model correctly classifies QUBO instances to the best solver 70% of the time and to the second-best solver 90% of the time. Furthermore, they proposed heuristics for solver parameter selection.

In this section, we will examine their work and extract valuable insights, such as explanatory variables for QUBO and the best ML models for quantum-classical algorithm selection. We will then outline how we can utilize this information for the ProvideQ toolbox.

### Motivation

The authors argue that regardless of the two different quantum paradigms for optimization problems - quantum annealing or quantum circuit models - the problem must first be written as a Quadratic Unconstrained Binary Optimization (QUBO) formulation involving only binary variables.

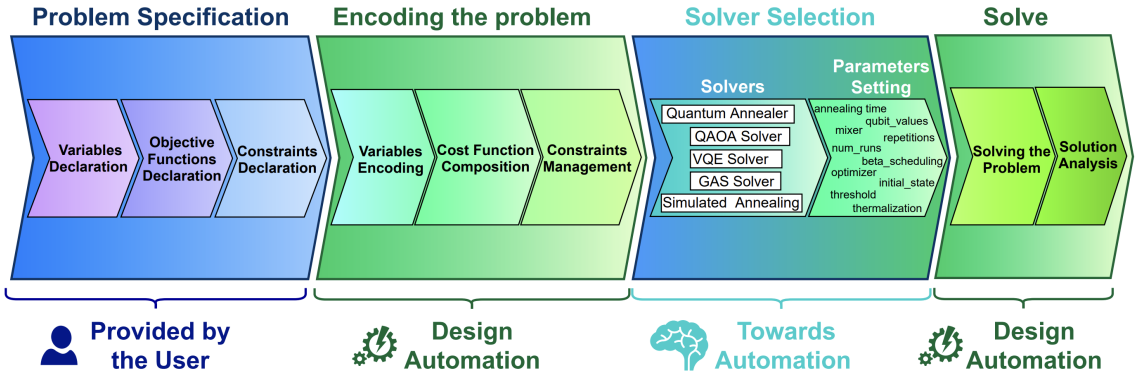



Figure 31: Quantum optimization flow [61].

Translating a problem instance into a QUBO instance, quantum-classical solver, and parameter selection requires domain expertise. Volpe et al. state that there are tools that help users encode the problem (see Figure 31) or interface with the solver. However, there are no tools that help users choose the best solver for a given use case.

First, the authors describe and compare quantum and quantum-classical algorithms: Quantum Annealer (QA), Quantum Approximate Optimization Algorithm (QAOA), Variational Quantum Eigensolver (VQE), and Grover Adaptive Search (GAS), as well as the classical counterpart of QA, Simulated Annealing (SA). Depending on the problem, they highlight that any of the aforementioned solvers can be better than the others in terms of solution quality. Due to the high cost of quantum hardware and the significant overhead of quantum simulations, the default approach of trying multiple solvers and parameters is infeasible.

For this reason, the authors developed an ML pipeline for the solver selection problem.

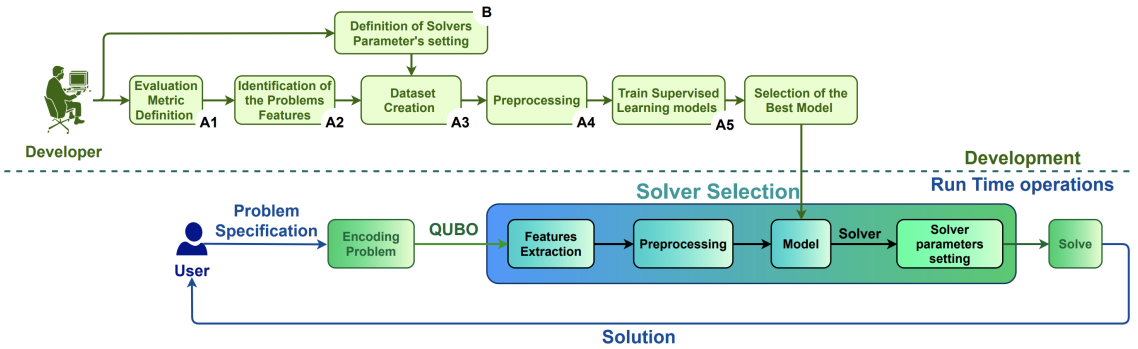



Figure 32: Workflow for implementation of the proposed predictive model and for its exploitation from the user perspective [61].

We now dive into their proposed solutions for steps A and B in as seen in Figure 32.

### Evaluation Metric ($A1$)

As an evaluation metric or criterion for choosing the best solver, Volpe et al. considered the time it takes to reach convergence and the quality of the solution. However, they argue that, due to the high solver runtime variance (which we outlined in section 5) and the inclusion of a classical simulated annealing algorithm that slows exponentially with the number of qubits in a circuit, solver runtime is not adequate.

The quality metric chosen by the authors is as follows:

$$F_s \;=\; -\alpha\, p_s \;+\; \beta\,(E_{\text{opt}} - E_{\text{ref}}) \;+\; \gamma\,(E_{\text{avg}} - E_{\text{ref}}) \;+\; \delta\, E_{\text{var}} \;-\; \eta\, p_v.$$

The idea is that, instead of considering standalone typical evaluation criteria, such as percentage of outcomes equal to the optimum, best achieved value, optimal reference value for the problem, average value obtained or percentage of solutions that satisfy these criteria, we could consider single formula that blends them together using chosen importance weights $\alpha, \beta, \gamma, \delta, \eta$.

The last issue that Volpe et al. addressed are occurrences in which multiple solvers achieve the same score. The authors considered an approach in which all tied solvers are correct and can be ranked as the best. For example, for instance i, the learning target could be a set of {QAOA, VQE}. However, since this increases complexity, the authors break ties deterministically by selecting one solver according to its ranking via a preference criterion based on the number of qubits used by the solver and a preference for quantum-classical solvers over purely classical ones, such as SA.

### Explanatory-Variables ($A2$)

Unlike the approach in To Quantum or Not to Quantum [40], the approach proposed by Volpe et al. extracts and identifies meaningful explanatory variables from the QUBO reformulation rather then original instance itself.

The authors identified nine features that capture the characteristics of the QUBO instances. They are as follows:

1. number of variables in the QUBO problem;
2. number of non-zero first-order QUBO coefficients ($a_i$);
3. number of non-zero second-order QUBO coefficients ($b_{ij}$);
4. average of non-zero first-order QUBO coefficients ($a_i$);
5. variance of non-zero first-order QUBO coefficients ($a_i$);
6. average of non-zero second-order QUBO coefficients ($b_{ij}$);
7. variance of non-zero second-order QUBO coefficients ($b_{ij}$);
8. average of all coefficients ($a_i$ and $b_{ij}$);
9. variance of all coefficients ($a_i$ and $b_{ij}$).

The explanatory variables were carefully selected so that their computation time would grow quadratically at most with the number of variables in the QUBO problem. This efficiency is essential because, once the model is trained and deployed for predictions, it must compute these explanatory variables for each QUBO instance before making a prediction. Ideally, this process should not take longer than the average time it takes to solve the instance with a solver.

**Dataset ($A3$)**

For training purposes, Volpe et al. gathered over 500 different QUBO instances that vary in size, density, coefficient range, and linear and nonlinear elements. For exact reference and descriptions, compare [61]. To determine the best solver for a given QUBO instance, they ran each instance more than 100 times on each solver to obtain reliable results despite their stochastic nature. Then, they labeled each instance with the best-performing solver using the performance function $F_s$.

The final dataset had the following label distribution: QAOA was the best solver in nearly 50% of cases, followed by QA and VQE, which were successful in $\approx$ 20% of cases. Lastly, GAS and SA found the optimum in $\approx$ 10% of the cases.

**Pre-processing ($A4$)**

To increase model stability, the authors first applied z-scoring to bring the values of the explanatory variables into the same range.

Furthermore, Volpe et al. reduced the dimensionality of the explanatory variables using unsupervised machine learning with principal component analysis (PCA) [30] and supervised machine learning with linear discriminant analysis (LDA) [57]. These approaches extract explanatory variables via variance maximization with principal components and projections onto a lower-dimensional subspace, respectively, while retaining crucial information in the data [61].

**Solver Parameter Settings ($B$)**

In this section authors propose how they choose (critical-) solver settings for each of the used solvers.

1. QA: the crucial parameter is called annealing time which scales with QUBO-size. Volpe et al. propose time-to-solution (TTS) metric in respect to problem size derived from plots.

2. QAOA: authors derive from plots the scaling function *reps*, which defines number of repetitions required for achieving good results.
3. VQE: has no critical parameters.
4. GAS: authors note that GAS has two crucial parameters, stop treshhold and number of qubits representing cost function values. Volpe et al. use two heuristics found in literature.
5. SA: similar to QA.

**Models ($A5$)**

The authors used the following models, which are available in Scikit-learn [46], for the best solver classification:

1. Ada Boost
2. Decision Tree
3. Gradient Boosting
4. k-nearest neighbors (KNN)
5. Logistic Regression
6. Naive Bayes
7. Neural Network
8. Random Forest
9. Support Vector Machine (SVM)
10. eXtreme Gradient Boosting (XGBoost)

Volpe et al. offer the pre-trained classifier on GitHub as part of the Munich Quantum Toolkit (MQT) (https://github.com/cda-tum/mqt-qao). However, they only offer it as a binary and do not provide the code used for training and evaluation. Furthermore, it is unclear if they deployed any hyperparameter search because they only report the hyperparameters used for the best-performing model.

**Results**

The results can be read from the table 17 below with the following metrics:

(i) accuracy determines how often did the model predict the right classifier

(ii) Top 2 denotes the relative frequency of predicting one of the top two solvers and

(iii) Average $p_s$ error is the average distance between the probability of achieving the optimal solution of the best solver and the predicted solver.

Table 17: Volpe et al. [61] Performance comparison across various classifiers, highlighting accuracy (Acc), the percentage of top two predictions, and the average error in the success probability (ps err). The best results are highlighted in bold and green. We note that we have adapted this table to not display the effects which explanatory-variables pre-processing had on the models, as it yielded no improvements for tree-based methods

| Model | **No Preproc.** | | |
|---|---|---|---|
| | Accuracy [%] | Top two [%] | $p_s$ err [%] |
| AdaBoost | 64.48 | 86.23 | 4.26 |
| Decision Tree | 68.65 | 87.50 | 3.22 |
| Gradient Boosting | 72.63 | 89.86 | 2.40 |
| KNN | 57.79 | 81.52 | 7.37 |
| Logistic Regression | 71.01 | 88.59 | 3.70 |
| Naive Bayes | 53.09 | 77.36 | 7.49 |
| Neural Network | 57.78 | 81.16 | 6.63 |
| Random Forest | **73.18** | **91.12** | **2.16** |
| SVM | 59.06 | 83.70 | 7.17 |
| XGBoost | 69.56 | 87.50 | 3.01 |

The results were achieved using four-fold cross-validation. Furthermore, to address the high class imbalance mentioned in the "Dataset" section, Volpe et al. implemented a stratification technique to ensure that each cross-validation fold maintained the same proportion of classes.

Tree-based models, such as Gradient Boosting, XGBoost, and most notably, Random Forest, achieved the best results. We note that, unlike in our previous RQs, Volpe et al. used classification instead of regression.

The authors acknowledge that, while these results are promising, they could be improved by enlarging the dataset, adding machine learning for parameter optimization, or developing a multi-label approach to manage parity cases instead of selection criteria.

#### Insights for ProvideQ

In this paragraph, we will discuss the valuable insights proposed by Volpe et al., which we could utilize in the ProvideQ toolbox in the future. Furthermore, we will propose changes to the Volpe et al. methodology, focusing on providing users with information and decision-making power instead of the black-box character forced by ML classifier models.

#### (1) QUBO explanatory-variables

The current ProvideQ prototype provides quantum solvers based on the QAOA and Grover's algorithms. Both require input in the form of a quadratic unconstrained binary optimization (QUBO) instance.

We believe the runtime and quality of the QAOA or Grover's algorithm solvers are influenced by the properties of the transformed QUBO rather than the explanatory variables of the original formulation, such as the traveling salesperson problem (TSP).

While multiple papers in the literature have engineered explanatory variables for classical domains (e.g., Hutter et al. [27]), as far as we know, only Volpe et al. have explicitly designed them for the QUBO domain.

In future work, we will explore the explanatory variables proposed by Volpe et al. for algorithms utilizing the QUBO format because we believe they will yield significant improvements over pre-transformed explanatory variables.

**(2) Choice of evaluation metric**

We will address the choice of evaluation metric, $A1$. Volpe et al. prepared the dataset for classification by ranking the best solver with the quality metric $F_s$. They also considered time to acquire convergence but decided against it due to high solver runtime variance and because comparing quantum-classical and purely classical algorithms is not representative due to current hardware limitations.

Although Volpe et al. considered time to acquire convergence as an evaluation metric for the labels in the classification dataset and not as a target variable for regression, we can draw parallels between their results and our results from RQ2.

The high solver runtime variance is consistent with our findings in RQ2. However, Volpe et al. indirectly offered a possible solution in the way they calculated their quality evaluation metric. Due to the stochastic nature of the solvers, they ran each algorithm a hundred times for each solver and then aggregated the optimal and average results into their quality function, $F_s$. We argue that the same methodology could be used for runtimes. Instead of the optimal runtime, we could define the quickest runtime and take the average across all runtimes, which would mitigate the high variance.

Furthermore, we agree that direct runtime comparisons of classical, hybrid quantum-classical, or simulated algorithms are problematic due to current qubit limits, the high cost of quantum hardware, and the exponential slowdown of simulated algorithms. However, we argue that runtime prediction carries significant meaning when evaluated across algorithms belonging to the same domain. This is evident in the ProvideQ running example for a VRP meta-solver strategy, where Qrisp QAOA is pitted against a D-Wave annealer. This is because it removes the black-box nature of the automation, ultimately empowering users to make informed decisions.

**(3) From classification to regression**

Building on the argument of the power of informed decisions, we propose exploring quality-metric regression instead of the classification of the best solver ($A5$). We acknowledge that focusing solely on runtime prediction misses cases where a longer runtime is worthwhile due to higher solution quality, as with approximation

algorithms. This would change the following in the pipeline proposed by Volpe et al.:

(i) In step A3, Volpe et al. ran each of their solvers 100 times for each QUBO instance to calculate the quality score $F_s$. Then, for each instance, they compared the $F_s$ scores that each algorithm delivered, picked the best one according to their preference criterion, and labeled the instance x with this solver.

We now present an exemplary dataset proposed by Volpe et al., which contains two samples, denoted by indices $x$ and $y$, with explanatory variables $e_1, e_2$ and their corresponding labels.

Table 18: Exemplary dataset table proposed by Volpe et al.

| Sample | **Features** | | **Label** |
|---|---|---|---|
| | $e_1$ | $e_2$ | Label |
| $x$ | 0.15 | 0.82 | $QAOA$ |
| $y$ | 0.47 | 0.36 | $VQE$ |

Next, in step $A5$, they train the supervised learning model, which, given samples with explanatory variables, tries to learn the algorithm labels.

We propose the following change: Instead of using the $F_s$ scores to determine the best label, we propose saving the normalized quality score $F_s$ as the target variable for regression instead of the algorithm label.

Careful readers will notice one major problem here: we lose information about which solver delivered the corresponding quality score, $F_s$.

One obvious solution is to add the labels to the explanatory variables. For the example with two solvers, QAOA and VQE, this would result in the following table:

Table 19: Exemplary dataset table for our first proposal

| Sample | **Features** | | **Label** | $F_s$ |
|---|---|---|---|---|
| | $e_1$ | $e_2$ | Label | $F_s$ |
| $x_1$ | 0.15 | 0.82 | $QAOA$ | 0.9 |
| $x_2$ | 0.15 | 0.82 | $VQE$ | 0.7 |
| $y_1$ | 0.47 | 0.36 | $QAOA$ | 0.3 |
| $y_2$ | 0.47 | 0.36 | $VQE$ | 0.6 |

In the above example, we would use the trained, deployed model as follows: Given a new QUBO instance, we would first calculate the explanatory variables for that instance. Then, we would run the model twice: once with the explanatory variable labeled as QAOA and once as VQE. Lastly, we would display the regressed $F_s$ values for the corresponding solvers, which would allow the user to make an informed decision.

We also considered a second option inspired by the evolution of the competition-winning SATZilla classifier, proposed by Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown [67]. Initially, in 2003 and 2004, SATZilla used a separate machine learning (ML) model for each solver to regress run times. To predict the best model for a given instance, they compared the run times predicted by each model for the corresponding solver and selected the one with the quickest prediction. Later, they moved from regression to an ensemble decision tree classifier. They trained one classifier for every pair of solvers in their portfolio to predict whether solver A or solver B was better. The results were then inferred by comparing the votes of the pairwise decision forest classifier and running the solver with the most votes. They decided to drop the runtime regression due to problems with censored runtimes in the training datasets. However, we note that the black-box nature of the classifier proposed by Xu et al. is appropriate for a competition setting, where the main goal is to pick the best solver.

We propose adapting the first variant proposed by Xu et al., but with the quality score, $F_s$, as the regressed value. This would result in a model for each algorithm, which can be derived from the dataset table depicted below.

Table 20: Exemplary datasets used for training of VQE and QAOA models

| Sample | **VQE Model** | | | **QAOA Model** | | |
|---|---|---|---|---|---|---|
| | $e_1$ | $e_2$ | $F_s$ | $e_1$ | $e_2$ | $F_s$ |
| $x$ | 0.15 | 0.82 | 0.70 | 0.15 | 0.82 | 0.90 |
| $y$ | 0.47 | 0.36 | 0.60 | 0.47 | 0.36 | 0.30 |

## 6.4 Discussion

Overall, we examined the ProvideQ structure and then reviewed the two most promising publications we found on quantum machine learning. Both thematized the classification of the best solver for a given instance. Moussa et al. examined the specific problem of MaxCut, comparing a purely classical approach with a hybrid quantum-classical approach. Volpe et al., on the other hand, addressed classifiers for all possible quantum problems that can be encoded as QUBO. We will now compare what we have learned about regression instead of classification, leaving the decision-making to the user and removing the black-box character of the automation process.

In a paper published by Moussa et al., the authors addressed the following question: Given an instance, can we determine whether it is worthwhile to run the costly quantum algorithm instead of its classical counterpart, modeled as classification for approximate classification? We gained two key insights. First, we found that using the performance of the classical algorithm, which can solve an instance in polynomial time, as an explanatory variable can provide valuable information about the possible performance improvements that the quantum-classical counterpart can achieve. This was novel because, up until that point, we had considered different algorithms completely independent of each other, only predicting their runtime as an information metric. In hindsight, we can draw parallels to the explanatory variable method

described by Hutter et al. for MIP, which we utilized in RQ1 and RQ2. Hutter et al. found that dynamic variables, such as statistics that can be extracted from a short, five-second pre-solve of an instance, can improve predictive power by approximately 20%. Moussa et al. treat the classical solver, which is computationally inexpensive, as a complementary explanatory variable for the best solver prediction. We could experiment with the explanatory variables proposed by Moussa et al. in ProvideQ for approximative optimization algorithms with classical and quantum subroutines, where the classical subroutines have a cheap runtime. The second insight concerns their model training. In our previous RQs, we did not consider the AutoML approach with TPOT, which seems worth exploring in future work.

Volpe et al. approached hybrid quantum-classical machine learning differently. They acknowledge that most problems solvable with NISQ-era algorithms must first be encoded in QUBO format, which hybrid quantum-classical solvers understand. Unlike Moussa et al., Volpe et al. did not focus on one specific problem, Max-Cut, but rather researched a supervised machine learning (ML) classifier for all problems that can be encoded as a QUBO. These problems can then be solved using quantum-classical algorithms and one classical simulation. To accomplish this, they derived significant explanatory variables for QUBO instances, offering insights into QUBO instances across various optimization domains, including networking and routing. Furthermore, they defined a solution-quality metric and used it to create a dataset for predicting which algorithm would yield the best results. Their findings align with our previous RQ findings that tree-based models perform best with tabular explanatory variables.

We will combine the findings from these two publications and outline the following plan for ProvideQ:

(1) First, we will implement solver run-time and quality prediction as regressions for all the classical and hybrid quantum-classical algorithms available in ProvideQ. For hybrid quantum-classical algorithms that use a QUBO formulation, we will use the explanatory variables provided by Volpe et al.

(2) For problem classes in which we implement both classical and quantum-classical algorithms and assuming the classical solver has a polynomial runtime, we will extend the explanatory variables for quantum-classical quality prediction with performance explanatory variables gained from the classical algorithm as proposed by Moussa et al.

(3) We will explore AutoML methods to see if we can improve ML models further, as opposed to the proposed XGBoost method.

Overall, we plan to provide ProvideQ users with metrics such as algorithm runtime and solution quality so they can make informed decisions about which solver to use for a given use case.

# 7 Related Work

### Machine Learning Models

During our research we have found many machine learning models, which we considered for implementation and evaluation of EPMs for run-time prediction. Notably, we considered Regression Trees [9], Random Forest [25], Gradient Boosted Trees [20] and lastly XGBoost [66]. Our contribution differs from these publications in the level of abstraction we use in our analysis. Instead of jumping directly into the deep mathematical details, we guide the readers through an abstract overview followed by an concrete, easy example. Lastly, we go into simplified mathematical details complemented with intuitive explanations for all formulas.

### Models evaluation for run-time prediction in ProvideQ

For our baseline algorithm runtime prediction, we considered the meta-study by Hutter et al. titled "Algorithm Runtime Prediction: Methods & Evaluation" [27]. We also reviewed other publications on EPM for runtime prediction, including Pouya et al.'s 2024 publication [47], which addressed the specific case of MIP for job shop scheduling. However, we questioned the methodology, finding it too specific to draw parallels to our general case in ProvideQ [16]. Other publications by Barry and Schumann titled "Strategies for Runtime Prediction and Mathematical Solvers Tuning" [4] extend the work of Hutter et al. by delving into specific details, such as improving synthetic data generation.

Our thesis distinguishes itself from the aforementioned publications by revisiting the decade-old Hutter et al. study. We reproduced the results and modernized the study to align with current state-of-the-art standards across three important topics: the machine learning model, the dataset, and the handling of censored run times.

To select the optimal model with the greatest potential, we examined the following studies: "XGBoost and Random Forest Algorithms: An In-Depth Analysis" by Fatima et al. [19] and "A Comparative Analysis of XGBoost" by Bentéjac et al. [5].

To select the most modern MIP dataset reflecting current instances across all domains, we referred to the MIPLIB2017 [21] library. To obtain the most reliable results, we hyper-tuned the models using OPTUNA [2], a state-of-the-art automated hyperparameter search library.

Furthermore, the study by Hutter et al. pointed out the problems with censored runtimes and a possible solution: Survival Analysis. To understand this concept in our domain we analyzed the oldest study that handled censored data for regression called "Linear regression with censored data" by Buckley et al [10]. Next, we moved on to a more modern study that handled survival analysis, called "Applied life data analysis" by Nelson et al. [41] and to concrete methods such as Cox-Regression, Kaplan-Meier and Accelerated-Failure-Time (AFT) [12, 32, 62]. We then implemented the AFT method with the XGBoost model provided by [65, 66].

Additionally, at the end of RQ2, we discovered a possible problem with the methodology: high variance in solver behavior for MIP. We reference a work by Hurley and O'Sullivan [26] that described this problem and proposed possible solutions: Statistical Regimes and Runtime Predictions.

**ML-based Performance Prediction Methods for Hybrid Quantum-Classical Solvers**

In RQ3, we transitioned from run-time prediction in the classical domain to the quantum-classical domain. Moussa et al. [40] described a machine learning approach that supports the selection of classical and quantum-classical solvers (Goemans-Williamson and QAOA) for Max-Cut instances. Another publication, titled "Predictive Approach for Selecting the Best Quantum Solver for an Optimization Problem" by Volpe et al. [61] implemented a machine learning model that selects the optimal classical or quantum-classical algorithm (QA, SA, QAOA, VQE, or GAS) for a given QUBO instance as part of the MQT Quantum Auto-Optimizer [64].

We discovered that none of the publications in this field addressed run-time prediction directly. Instead, they handle black-box automation with machine learning, which proposes the best algorithms for a given problem instance through classification.

Inspired by these two publications, our thesis outlined an improved pipeline for ProvideQ. Our proposal is: instead of *letting* the model make the decision, we *suggest* using machine learning to help users make the decision. Our proposal outlines a switch from classification to regression of run-time and solution-quality metrics for quantum-classical subroutines, which is novel in this domain.

# 8 Conclusion

Our goal in this thesis was to analyze how we can empower the ProvideQ toolbox users to select the best Meta-Solver strategy for their use-case, without the need for domaine expertise. To this end, we researched empirical performance models (EPMs) that predict runtime for both classical and quantum-classical solvers.

This thesis consists of four major contributions: (1) First, we began by presenting in an intuitive way and explaining in detail the machine learning models relevant for the solver runtime prediction: regression trees, random forests, gradient boosting and XGBoost. In addition, we describe all the related concepts used in the machine learning pipeline - explanatory variables, handling of data, and evaluation. Furthermore we analyzed the literature that compares the presented models and we justified the most promising model, XGBoost, that we have used for for subsequent improvements. This fundament allowed us to make the subsequent design choices understandable.

(2) Second, we revisited the meta-study of Hutter et al. in RQ1. We introduced and summarized their EPM pipeline by describing the datasets used, how algorithm runtimes were handled, the relevant explanatory variables used, and how different ML models were evaluated. Furthermore, we updated their legacy MATLAB/C code and reproduced their experimental results on MIP a decade later. Our reproduction of the results on the heterogeneous BIGMIX dataset with CPLEX runtimes matched the reported results. We were able to slightly improve upon them through code sanitization alone. We observed the similar performance of the Random Forest (ML) model and confirmed Hutter et al.'s conclusion that it is a powerful model for runtime prediction on heterogeneous datasets. Notably, this conclusion follows the dynamic explanatory variables Hutter et al. introduced in their thesis for the first time in all available literature.

(3) Third, we moved from experiment reproduction to modernization. We transferred part of the experiment from MATLAB to Python, improved the RF with modern hyperparameter optimization, and introduced XGBoost, the state-of-the-art model for tabular data. Furthermore, we replaced the outdated BIGMIX dataset with the more recent MIPLIB2017 dataset. We updated the XGBoost pipeline on MIPLIB: we recalculated decade-old run times with a modern CPLEX version, extracted new explanatory variables from the MIPLIB dataset, and evaluated the model against the baseline. Our findings are as follows: (i) the exact (*absolute*) run-time prediction of modern MIP instances is difficult. However, we discovered that the model performs well in the *relative* ordering of the run times in regard to their difficulty. Second, we found that censored data matters. Treating solver runtime timeouts as 3600 seconds biases the model to underpredict the runtimes for hard MIP instances. Our XGBoost-AFT survival analysis variant uses censored intervals instead, improving the detection of long-running or non-terminating instances by providing more conservative runtime predictions for MIP instances with longer recorded runtimes, while slightly sacrificing the overall predictive performance. Based on these results, we derived a heuristic that flags possible timeouts while still providing relative run times for comparison across subroutines.

(4) Lastly, we researched the application of ML-based performance prediction methods for hybrid quantum-classical solvers. We analyzed two relevant publications and gained insights into explanatory variables for QUBO problem formulations used by NISQ-era quantum-classical solvers, as well as insights into solver-selection classifiers opposed to run-time prediction regressors. We proposed an outline how machine learning can be implemented into ProvideQ for hybrid quantum-classical problems with the emphasis on letting the users make informed decision. To this end, we proposed a further prediction value, in addition to the run-time value used thus far: solution-quality. This allows for a fair comparison between classical and quantum-classical algorithms.

Overall, we created and shared a modern EPM pipeline for MIP run-time prediction, which is ready to be implemented into ProvideQ. However, we are pragmatic with our results. We argue that EPMs are a good guidance for sub-routine selection, however, the predicted solver run-times in seconds should be treated with caution.

**Future Work**

In this thesis, we explored many complex branches of EPMs. Building on this starting block, we plan to explore the following topics in future work:

1. We want to extend the developed vertical slice on MIP, with EPMs for other sub-routines available in ProvideQ: TSP/VRP, Knapsack, Max-Cut, SAT and QUBO.

2. Analyze the issue of high solver-runtime variability on the same instances due to heuristics.

3. Extend the run-time prediction with solution-quality prediction for approximation combinatorial optimization algorithms.

4. Revisit AFT Survival Analysis on XGBoost in detail, the most promising approach in our thesis.

# Acknowledgments

We would like to thank Frederik Fiand for the early consultation regarding this thesis and for pointing out the potential problems of solver variability, and Hannah Bakker for the consultations and the idea of using ML models on homogeneous datasets for each problem domain in MIP, instead od generalized ones.

# References

[1] Kiyan Ahmadizadeh, Bistra Dilkina, Carla P. Gomes, and Ashish Sabharwal. An empirical study of optimization for maximizing diffusion in networks. In David Cohen, editor, *Principles and Practice of Constraint Programming – CP 2010*, pages 514–521, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[2] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2019.

[3] Giorgio Ausiello, M. Protasi, A. Marchetti-Spaccamela, G. Gambosi, P. Crescenzi, and V. Kann. *Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties*. Springer-Verlag, Berlin, Heidelberg, 1st edition, 1999.

[4] Michael Barry and Michael Schumacher. Strategies for runtime prediction and mathematical solvers tuning. *Proceedings of the 11th International Conference on Agents and Artificial Intelligence - (Volume 2)*, (CONFERENCE):Pp. 669–676.

[5] Candice Bentéjac, Anna Csörgő, and Gonzalo Martínez-Muñoz. A comparative analysis of xgboost, 11 2019.

[6] James Bergstra, Daniel Yamins, and David Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In Sanjoy Dasgupta and David McAllester, editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 115–123, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR.

[7] James Bergstra, Daniel Yamins, and David D. Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 115–123, Atlanta, Georgia, USA, 17–19 June 2013. PMLR.

[8] Dimitris Bertsimas and John N. Tsitsiklis. *Introduction to Linear Optimization*. Athena Scientific, 1997.

[9] Leo Breiman, Jerome H. Friedman, Richard A. Olshen, and Charles J. Stone. *Classification and Regression Trees*. Chapman & Hall / CRC, 1984.

[10] JONATHAN BUCKLEY and IAN JAMES. Linear regression with censored data. *Biometrika*, 66(3):429–436, 12 1979.

[11] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 785–794. ACM, August 2016.

[12] D. R. Cox. Regression models and life-tables. *Journal of the Royal Statistical Society. Series B (Methodological)*, 34(2):187–220, 1972.

[13] D. R. Cox. Partial likelihood. *Biometrika*, 62(2):269–276, 1975.

[14] Pedro Domingos. A few useful things to know about machine learning. *Commun. ACM*, 55(10):78–87, October 2012.

[15] Domenik Eichhorn, Maximilian Schweikart, Nick Poser, Frederik Fiand, Benedikt Poggel, and Jeanette Miriam Lorenz. Hybrid meta-solving for practical quantum computing. In *2024 IEEE International Conference on Quantum Computing and Engineering (QCE)*, volume 01, pages 421–431, 2024.

[16] Domenik Eichhorn, Maximilian Schweikart, Nick Poser, Frederik Fiand, Benedikt Poggel, and Jeanette Miriam Lorenz. Hybrid meta-solving for practical quantum computing. In *2024 IEEE International Conference on Quantum Computing and Engineering (QCE)*, volume 01, pages 421–431, 2024.

[17] Michael Falkenthal, Christoph Krieger, Felix Paul, Sebastian Wagner, and Michael Wurster. Planqk—platform and ecosystem for quantum applications. *KI-Künstliche Intelligenz*, pages 1–7, 2024.

[18] Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. A quantum approximate optimization algorithm, 2014.

[19] Sana Fatima, Ayan Hussain, Sohaib Amir, Syed Haseeb Ahmed, and Syed Aslam. Xgboost and random forest algorithms: An in depth analysis. *Pakistan Journal of Scientific Research*, 3:26–31, 10 2023.

[20] Jerome Friedman. Greedy function approximation: A gradient boosting machine. *The Annals of Statistics*, 29, 11 2000.

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

[22] Michel X. Goemans and David P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *J. ACM*, 42(6):1115–1145, November 1995.

[23] Carla P. Gomes, Willem-Jan van Hoeve, and Ashish Sabharwal. Connections in networks: A hybrid approach. In Laurent Perron and Michael A. Trick, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 303–307, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[24] G. Highleyman. A method for train–test data splitting. 1962. Earliest reference to the holdout method.

[25] Tin Kam Ho. Random decision forests. In *Proceedings of 3rd international conference on document analysis and recognition*, volume 1, pages 278–282. IEEE, 1995.

[26] Barry Hurley and Barry O'Sullivan. Statistical regimes and runtime prediction. In Qiang Yang and Michael J. Wooldridge, editors, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence (IJCAI 2015)*, pages 318–324. AAAI Press, 2015.

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

[28] IBM. What is machine learning (ml)? https://www.ibm.com/think/topics/machine-learning. Accessed: 2025-08-19.

[29] Ali Javadi-Abhari, Matthew Treinish, Kevin Krsulich, Christopher J. Wood, Jake Lishman, Julien Gacon, Simon Martiel, Paul D. Nation, Lev S. Bishop, Andrew W. Cross, Blake R. Johnson, and Jay M. Gambetta. Quantum computing with Qiskit, 2024.

[30] Ian T. Jolliffe and Jorge Cadima. Principal component analysis: a review and recent developments. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 374(2065):20150202, 2016.

[31] D. R. Jones, C. D. Perttunen, and B. E. Stuckman. Lipschitzian optimization without the lipschitz constant. *Journal of Optimization Theory and Applications*, 79(1):157–181, 1993.

[32] Edward L. Kaplan and Paul Meier. Nonparametric estimation from incomplete observations. *Journal of the American Statistical Association*, 53(282):457–481, 1958.

[33] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: a highly efficient gradient boosting decision tree. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 3149–3157, Red Hook, NY, USA, 2017. Curran Associates Inc.

[34] Leonid G. Khachiyan. A polynomial algorithm in linear programming. *Soviet Mathematics Doklady*, 20:191–194, 1979.

[35] Victor Klee and George J. Minty. How good is the simplex algorithm? In Oved Shisha, editor, *Inequalities III*, pages 159–175. Academic Press, 1972.

[36] Thorsten Koch, Timo Berthold, Jaap Pedersen, and Charlie Vanaret. Progress in mathematical programming solvers from 2001 to 2020. *EURO Journal on Computational Optimization*, 10:100031, 2022.

[37] Thorsten Koch, Timo Berthold, Jaap Pedersen, and Charlie Vanaret. Progress in mathematical programming solvers from 2001 to 2020. *EURO Journal on Computational Optimization*, 10:100031, 2022.

[38] Ron Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'95, page 1137–1143, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[39] Kevin Leyton-Brown, Mark Pearson, and Yoav Shoham. Towards a universal test suite for combinatorial auction algorithms. In *Proceedings of the 2nd ACM Conference on Electronic Commerce (EC '00)*, pages 66–76, Minneapolis, MN, USA, 2000. ACM.

[40] Charles Moussa, Henri Calandra, and Vedran Dunjko. To quantum or not to quantum: towards algorithm selection in near-term quantum optimization. *Quantum Science and Technology*, 5(4):044009, October 2020.

[41] Wayne B Nelson. *Applied life data analysis*. John Wiley & Sons, 2003.

[42] George L Nemhauser and Laurence A Wolsey. *Integer and Combinatorial Optimization*. John Wiley & Sons, 1988.

[43] Gerhard Neumann and Benjamin Schäfer. Grundlagen der künstlichen intelligenz. Lecture notes, Winter Semester 2024/2025, 2024.

[44] NVIDIA Corporation. What is scikit-learn?

[45] Randal S. Olson, Nathan Bartley, Ryan J. Urbanowicz, and Jason H. Moore. Evaluation of a tree-based pipeline optimization tool for automating data science. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, GECCO '16, page 485–492, New York, NY, USA, 2016. Association for Computing Machinery.

[46] 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.

[47] Shaheen Pouya, Oguz Toragay, and Mehrdad Mohammadi. *Predicting the Solution Time for Optimization Problems Using Machine Learning: Case of Job Shop Scheduling Problem*, pages 450–465. 02 2024.

[48] Ruslan Shaydulin, Changhao Li, Shouvanik Chakrabarti, Matthew DeCross, Dylan Herman, Niraj Kumar, Jeffrey Larson, Danylo Lykov, Pierre Minssen, Yue Sun, Yuri Alexeev, Joan M. Dreiling, John P. Gaebler, Thomas M. Gatterman, Justin A. Gerber, Kevin Gilmore, Dan Gresh, Nathan Hewitt, Chandler V. Horst, Shaohan Hu, Jacob Johansen, Mitchell Matheny, Tanner Mengle, Michael Mills, Steven A. Moses, Brian Neyenhuis, Peter Siegfried, Romina Yalovetzky, and Marco Pistoia. Evidence of scaling advantage for the quantum approximate optimization algorithm on a classically intractable problem. *Science Advances*, 10(22):eadm6761, 2024.

[49] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, October 1997.

[50] Daniel A. Spielman and Shang-Hua Teng. Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time. *Journal of the ACM*, 51(3):385–463, 2004.

[51] J. Starmer. *The StatQuest Illustrated Guide to Machine Learning!!!: Master the Concepts, One Full-Color Picture at a Time, from the Basics All the Way to Neural Networks. BAM!* Packt Publishing, Limited, 2022.

[52] Joshua (StatQuest) Starmer. Statquest: Gradient boost & xgboost (playlist). YouTube Playlist, 2025. Covers Gradient Boosting and XGBoost, including mathematical details.

[53] Joshua (StatQuest) Starmer. Statquest: Random forests (playlist). YouTube Playlist, 2025. Explains Random Forest building, usage, evaluation.

[54] Deanne Taylor. Representative kaplan–meier survival plot (km_plot.jpg). https://commons.wikimedia.org/w/index.php?title=File:Km_plot.jpg&oldid=455210939, 2005. Wikimedia Commons. Public domain. Accessed 2025-08-21.

[55] DATAtab Team. Cox regression, March 2025. Tutorial.

[56] DATAtab Team. Kaplan–meier curve, March 2025. Tutorial.

[57] Alaa Tharwat, Tarek Gaber, Abdelhameed Ibrahim, and Aboul Ella Hassanien. Linear discriminant analysis: A detailed tutorial. *AI Communications*, 30(2):169–190, 2017.

[58] UBC Department of Computer Science. Lecture 13: Feature engineering and feature selection. CPSC 330: Applied Machine Learning, University of British Columbia, 2023. Online lecture notes; includes Andrew Ng quote "coming up with features is difficult, time-consuming, requires expert knowledge. 'Applied machine learning' is basically feature engineering.".

[59] UC Berkeley School of Information. What is machine learning (ml)? https://ischoolonline.berkeley.edu/blog/what-is-machine-learning/, 2022. Accessed: 2025-08-19.

[60] D. Volpe, N. Quetschlich, M. Graziano, G. Turvani, and R. Wille. Towards an Automatic Framework for Solving Optimization Problems with Quantum Computers. In *IEEE International Conference on Quantum Software (QSW)*, 2024.

[61] Deborah Volpe, Nils Quetschlich, Mariagrazia Graziano, Giovanna Turvani, and Robert Wille. A predictive approach for selecting the best quantum solver for an optimization problem, 2024.

[62] L. J. Wei. The accelerated failure time model: A useful alternative to the cox regression model in survival analysis. *Statistics in Medicine*, 11(14–15):1871–1879, 1992.

[63] Wikipedia contributors. Standard score. https://en.wikipedia.org/w/index.php?title=Standard_score&oldid=1304956397, 2025. Accessed: 2025-08-21.

[64] Robert Wille, Lucas Berent, Tobias Forster, Jagatheesan Kunasaikaran, Kevin Mato, Tom Peham, Nils Quetschlich, Damian Rovara, Aaron Sander, Ludwig Schmid, Daniel Schönberger, Yannick Stade, and Lukas Burgholzer. The

mqt handbook: A summary of design automation tools and software for quantum computing. In *2024 IEEE International Conference on Quantum Software (QSW)*, page 1–8. IEEE, July 2024.

[65] XGBoost Contributors. Accelerated failure time (aft) survival analysis. https://xgboost.readthedocs.io/en/latest/tutorials/aft_survival_analysis.html, 2024. Accessed: 2025-07-28.

[66] XGBoost Contributors. *XGBoost Documentation*. Read the Docs, 2025. Accessed: 2025-08-17.

[67] Lin Xu, Frank Hutter, Jonathan Shen, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla2012: Improved algorithm selection based on cost-sensitive classification models. In *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, pages 57–58, Helsinki, Finland, 2012. Department of Computer Science Report Series B, No. B-2012-2, University of Helsinki.