Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science

Hyperparameter Tunning for Local Search Algorithms

Master's thesis

Bc. Lukáš Haberzettl

Study Programme: Artificial Intelligence
Field of Study: Open Informatics
Supervisor: Ing. Jiří Čermák, Ph.D.

Prague, May 2020

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Haberzettl Lukáš** Personal ID number: **435465**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Computer Science**

Study program: **Open Informatics**

Specialisation: **Artificial Intelligence**

II. Master’s thesis details

Master’s thesis title in English:

Hyperparameter Tunning for Local Search Algorithms

Master’s thesis title in Czech:

Optimalizace hyperparametrů pro algoritmy lokálního prohledávání

Guidelines:

Local search algorithms are common tools applied to real-world optimization tasks, where the size or complexity of the solved problem
prevents the use of optimal algorithms. Most of the available open-source and commercial tools allow extensive control of the parameters
guiding the search with very little information about their meaning and interconnection. Hence automatic tools for such parameter
optimization can greatly improve the performance of such algorithms.

- Exploration of methods suitable for optimization of parameters of black-box functions (the achieved solution quality of local
search algorithms).
- Implementation of Bayesian optimization and possibly other promising approaches for parameter tuning of the black box functions.
- Adaptation of the general approach for black-box parameter optimization to the domain of local search, e.g., speed-up through
estimation of achievable solution quality for given parameters using properties of the local search algorithm.
- Experimental evaluation of the selected methods against commonly used approaches such as random or grid search parameter
optimization using popular open-source planners (e.g., Optaplanner, Jsprit) on common benchmark domains.

Bibliography / sources:

Bergstra, J. and Bengio Y. – Random Search for Hyper-Parameter Optimization – Journal of Machine Learning Research, 2012
Snoek, J., Larochelle, H., and Adams, R.P. – Practical Bayesian Optimization of Machine Learning Algorithms – Advances in Neural Information Processing Systems, 2012
Balandat, M., and colleagues – BoTorch: Programmable Bayesian Optimizationin PyTorch – arXiv e-prints, 2019
Gendreau, M. and Potvin, J.-Y. – Handbook of Metaheuristics – Springer, 2019
Falkner, S., Klein, A., Hutter, F. 2018. BOHB: Robust and Efficient HyperparameterOptimization at Scale. arXiv e-prints arXiv:1807.01774

© ČVUT v Praze, Design: ČVUT v Praze, VIC

Name and workplace of master's thesis supervisor:

Ing. Jiří Čermák, Ph.D., Artificial Intelligence Center, FEE

Name and workplace of second master's thesis supervisor or consultant:

Date of master's thesis assignment: **17.02.2020** Deadline for master's thesis submission: **22.05.2020**

Assignment valid until: **19.02.2022**

Ing. Jiří Čermák, Ph.D. Supervisor's signature	Head of department's signature	prof. Mgr. Petr Páta, Ph.D. Dean's signature

III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student's signature

© ČVUT v Praze, Design: ČVUT v Praze, VIC

Acknowledgements

First, I would like to thank my thesis supervisor Jiří Čermák, whose valuable expertise gave me the right direction when working on the challenges of this work.
I must also express my gratitude to Petr Eichler for his valuable comments and advice that increased the quality of this thesis.
Finally, I would like to thank to my family, who provided me with support and encouragement throughout my years of study.

Author statement

I hereby declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, 22th May, 2020

....

author

Abstrakt

Algoritmy lokálního prohledávání jsou dnes široce používaným nástrojem pro řešení komplexních optimalizačních problémů, pro které není možné s dostupnými prostředky nalézt optimální řešení. Vnitřní strategie těchto algoritmů jsou často konfigurovatelné prostřednictvím hyperparametrů, které mohou významně ovlivnit jejich běh a efektivitu. Správné nastavení těchto parametrů je však často komplikované nedostatkem informací o jejich účinku a vzájemných závislostech. Vysoká časová náročnost běhu také zamezuje použití běžných metod pro optimalizaci hyperparametrů, zvláště v případech vysoké dimenzionality, kde prostor prohledávaných konfigurací je příliš velký.
V této práci studujeme metodu Bayesovské optimalizace jako vhodného kandidáta pro optimalizaci hyperparametrů časově náročných funkcí a obhajujeme její aplikovatelnost pro doménu algoritmů lokálního prohledávání. Pro řešení problému navrhujeme modifikaci Bayesovské optimalizace využívající informaci běhu algoritmů pro zrychlení optimalizačního procesu. Naše modifikovaná metoda prokázala rychlejší konvergenci než klasický Bayesovský přístup ve většině provedených experimentů s nalezením podobných nebo lepších hodnot hyperparametrů.

Keywords: Algoritmy lokálního prohledávání, Optimalizace hyperparametrů, Bayesovská optimalizace, Optimalizace

Abstract

Local search algorithms are widely used instruments for the complex optimization tasks, where the problem of finding the optimal solution is infeasible. Commonly, the search strategies of such algorithms can be controlled by a set of hyperparameters that can significantly affect their performance. However, the information about their meaning is often unclear or completely hidden, and identifying optimal values for hyperparameters can be a complicated and time-consuming task. Hence, an automated tool for such hyperparameter tunning can lead to significant performance improvement for the algorithms.

In this work, we study the Bayesian optimization as a state-of-the-art tool for automated hyperparameter tunning of expensive black-box functions, and uphold its usefulness when applied to local search algorithms. We propose a modification of the Bayesian approach that adapts to the domain of local search algorithms to speed up optimization process. In most of the performed experiments, our modified method proved to be faster than classic Bayesian optimization with similar or better hyperparameter configurations discovered.

Keywords: Local search algorithms, Hyperparameter tunning, Bayesian optimization, Optimization

Contents

List of Tables **xv**

List of Figures **xvi**

1 Introduction **1**
1.1 Contribution 1
1.2 Thesis outline 2

2 Existing methods for hyperparameter optimization **3**
2.1 Local search algorithms 3
2.2 Bayesian optimization 4
2.2.1 Optimization objective 4
2.2.2 Components 5
2.2.3 Optimization algorithm 8
2.2.4 Surrogate initialization 8
2.3 Other methods 9
2.3.1 Grid search 9
2.3.2 Random search 9
2.4 Comparison 10

3 Bayesian Optimization and Local Search Algorithms **11**
3.1 Terms 11
3.2 Selection of local search algorithms 12
3.2.1 TASP 12
3.2.2 Jsprit 13
3.2.3 OptaPlanner 13
3.2.4 VRPTW 14
3.3 Bayesian optimization analysis 15
3.3.1 Implementation 15
3.3.2 Approach verification 16
3.4 Domain adaptation 18
3.4.1 Solver progress curve 18
3.4.2 Early stopping 20
3.4.3 ARIMA model 21
3.4.4 Random Forest regression 23
3.4.5 Comparison 27

4 Algorithms **29**
4.1 Overall architecture . . . 29
4.1.1 Solver specification . . . 31
4.1.2 Early Stop policy overview . . . 32
4.2 Evaluator . . . 32
4.2.1 Solver evaluation . . . 32
4.2.2 Time series extraction . . . 33
4.3 Predictor . . . 35
4.3.1 Prediction with time window . . . 35
4.4 Controller . . . 36
4.5 Optimizer . . . 39
4.5.1 Initialization trials . . . 39
4.5.2 Bayesian optimization . . . 39

5 Experiments **41**
5.1 Early Stop policy analysis . . . 41
5.1.1 TASP . . . 42
5.1.2 Jsprit . . . 43
5.1.3 OptaPlanner . . . 43
5.1.4 Performance results . . . 46
5.2 Comparison with Bayesian optimization . . . 47
5.3 Generalization to unknown problem instances . . . 49

6 Conclusion **51**
6.1 Future work . . . 52

Bibliography **53**

A Attached files **56**

List of Tables

3.1 Hyperparameters definition for TASP 13
3.2 Hyperparameters definition for Jsprit 14
3.3 Hyperparameters definition for OptaPlanner 14
3.4 Selected set of problem instances 15
3.5 Fixed setting for local search algorithms 16
3.6 Example of time-series data 20
3.7 Feature set for Random Forest regression 26
3.8 Random Forest and ARIMA comparison 27

4.1 Parameter structure 31

5.1 Early Stop policy experiment settings 42
5.2 Controller experiment configuration 42
5.3 Early stop policy performance results 47
5.4 Solver configurations used in experiments 47
5.5 Fixed settings for generalization experiments 49
5.6 Results of generalization experiment 50

List of Figures

2.1 GP as surrogate model and NEI as acquisition function shown at one iteration . . . 7
2.2 Grid search and Random search comparison . . . 10

3.1 Bayesian optimization vs Random search . . . 17
3.2 Local search algorithm progress . . . 19
3.3 ARIMA model prediction for TASP time-series data . . . 23
3.4 Random Forest regression for TASP time-series data . . . 26
3.5 Random Forest regression for Jsprit time-series data . . . 26
3.6 Random Forest regression for OptaPlanner time-series data . . . 27

4.1 Optimization framework architecture . . . 30
4.2 Solver progress curve observation . . . 34
4.3 Controller lifecycle . . . 38

5.1 Early Stop policy analysis - TASP . . . 44
5.2 Early Stop policy analysis - Jsprit . . . 45
5.3 Early Stop policy analysis - OptaPlanner . . . 46
5.4 Comparison of ESBO and classic Bayesian optimization . . . 48
5.5 Comparison of optimization methods on unknown data . . . 50

Chapter 1

Introduction

In many optimization tasks, the use of optimal algorithms is not possible due to the size of the search space[1] or the complexity of the problem. In practice, local search algorithms are used to solve such problems, searching for the locally optimal solution. Local search algorithms are metaheuristic methods, using the approach of generic optimization of the initial, imperfect solution by searching in the neighborhoods and maximizing the criterion among the set of candidate solutions. Nowadays, the application of these algorithms is facilitated by software frameworks, providing the toolkits to solve these computationally hard optimization problems such as Traveling Salesman Problem (TSP), Vehicle Routing Problem (VRP), and many others. These frameworks are commonly configurable with a set of hyperparameters that can affect the overall performance of the algorithm. However, the lack of information about their meaning and interconnection often makes it hard to select parameters correctly. In practice, values of these parameters are selected either manually, by empirical observations, or by using conventional hyperparameter tunning methods like grid search or random search. However, due to the time complexity of the local search algorithms, these methods are typically not able to find sufficient hyperparameter configuration in a reasonable time frame, especially for high dimensions of hyperparameter space.

1.1 Contribution

In this work, the Bayesian optimization method is studied as a state-of-the-art optimization tool suitable for such hyperparameter optimization. We compare the Bayesian approach with standard hyperparameter optimization methods and validate its applicability for our case. Due to our research, we present the modification of the Bayesian

[1] Space of all possible solutions for the given problem

approach, adapted to the domain of the local search algorithms, as a framework for automated hyperparameter tunning. Our solution uses domain knowledge to speed up the underlying Bayesian optimization by stopping the unpromising evaluations of the objective function (local search algorithm) earlier in the execution process. Consequently, the optimization is able to evaluate more hyperparameter configurations in order to find the high-quality one. Implementation of our method is tested on the selection of local search algorithms and problem instances. Compared with the classic Bayesian approach, our method results in faster convergence to the high-quality configuration in most cases.

1.2 Thesis outline

The thesis has the following structure:

- **Chapter 2** describes and compares the subset of existing hyperparameter optimization approaches. Studied methods are grid search, random search, and Bayesian optimization.

- **Chapter 3** is dedicated to the extensive research of the Bayesian optimization applicability to the domain of local search algorithms. Also, the possible modification of the method is discussed and tested by experiments.

- **Chapter 4** presents our Early Stop Bayesian optimization framework. Individual components of the framework are explained along with the core concepts of our modification of the underlying Bayesian method.

- **Chapter 5** describes the results of experiments, that were performed to test our approach. It presents the experiments that were done to compare our method to the classic Bayesian optimization and validate its ability to generalize to the unknown problem instances of the local search algorithms.

- **Chapter 6** provides the conclusion of our work and results, and discuss the possibilities for future work on our approach.

Chapter 2

Existing methods for hyperparameter optimization

In this chapter, we provide an overview of existing optimization methods and discuss their potential for hyperparameter optimization (HPO) of the local search algorithms. Typically, it may take a long time for the local search algorithm to find a high-quality solution. This factor limits the number of hyperparameter configurations that can be tried out by the HPO method and makes the problem of searching through all possible combinations infeasible (Especially for the higher dimensions of hyperparameter space). Therefore, it is required for the optimization method to explore the search space of parameters effectively, minimizing the number of function (local search algorithm) calls needed for finding optimal hyperparameter configuration.

Given these requirements, we study the Bayesian Optimization method as a promising candidate for the HPO task and discuss its possible benefits over other optimization methods.

In Section 2.1, we provide overview of the local search algorithms. In Section 2.2, we describe the principle of Bayesian Optimization and uphold its possible application for the hyperparameter optimization of the local search algorithms. We then present other optimization methods in Section 2.3 and compare them with Bayesian optimization in Section 2.4.

2.1 Local search algorithms

Local search algorithms are used for the optimization of such problems, for which finding the optimal solution is infeasible. Thus, rather than finding an optimal solution, local search algorithms introduce a generic approach of searching through the space

of candidate solutions, iteratively selecting better candidates until the near-optimal solution is found. Algorithms typically provide a set of hyperparameters that are used to control the search process and can influence the performance significantly. In practice, hyperparameters found for one subset of problem instances generalizes well on the other, unknown instances. Hence, it is easy to justify using some extra time (computational) resources to optimize the hyperparameters for the algorithm.

2.2 Bayesian optimization

Bayesian optimization [1] is the global optimization method, which has shown its success with several benchmark functions, outperforming other state-of-the-art global optimization approaches [2]. Over the last years, this approach has emerged as an efficient optimization framework for machine learning models [3], where the well-chosen configuration of hyperparameters can affect the performance significantly.
This method is well suited for optimizing parameters of expensive black-box functions, where there is only objective value for the function available, the execution of the function is expensive, and we have no information about the gradient. Since local search algorithms typically satisfy these properties, we see the potential in this optimization approach as an HPO tool for such domain.
As the Bayesian optimization method treats objective as a black-box function, the only information retrieved from the function is the objective value. The method uses these observations of the objective value from the function calls to maintain the posterior distribution for the objective function. To pick the next input for the evaluation, the posterior distribution is used to define the acquisition function, which defines how promising it is to evaluate the objective function for a given point (in our case, configuration of hyperparameters). By optimizing the acquisition function, most promising point for the next evaluation is selected. In the following subsections, we describe individual components of the Bayesian optimization framework more extensively.

2.2.1 Optimization objective

In our context, we present the black-box objective function as a local search algorithm being evaluated for the fixed set of problem instances with given iteration limit[1]. The algorithm takes set of d hyperparameters $x \in \mathbb{R}^d$ as an input, which can affect the performance of the algorithm and can lead to different costs of the best solution found by the algorithm.

[1]Number of iterations that the local search algorithm is allowed to perform to find the solution

Given this, black-box objective function for optimization is defined as

$$f : \mathbb{R}^d \rightarrow \mathbb{R} \tag{2.1}$$

where $d \in \mathbb{N}$ is the dimension of the function input. The optimization problem is then defined as minimization of the objective value of the black-box function.

$$min_{\boldsymbol{x} \in \mathbb{R}^d} f(\boldsymbol{x}) \tag{2.2}$$

2.2.2 Components

Two main components are used by the Bayesian optimization method to optimize the objective. Surrogate model is used to create a probabilistic belief about the function, which is then exploited by the acquisition function to determine the next input for the objective function.

Surrogate model

Surrogate model is Bayesian statistical model used for modeling the probability distribution based on the previously made observations of $f(\boldsymbol{x})$. This fact is what distinguishes Bayesian optimization from many other methods. Instead of relying only on local gradient, method uses all the information observed from the previous evaluations of $f(\boldsymbol{x})$. This fact is a key factor for finding a minimum of the objective function in relatively few evaluations.
One of the statistical models widely used as a surrogate for Bayesian optimization, is Gaussian process [4]. It is also used in our experiments in Chapter 3.
A Gaussian process is a random process for which any point $\boldsymbol{x} \in \mathbb{R}^d$ has assigned a random variable $g(\boldsymbol{x})$. For a finite set of these variables, their joint distribution is itself Gaussian and defines a prior over functions.

$$p(\boldsymbol{g}|\boldsymbol{X}) = \mathcal{N}(\boldsymbol{g}|\boldsymbol{\mu}, \boldsymbol{K}) \tag{2.3}$$

where $\boldsymbol{g} = (g(\boldsymbol{x}_1), ..., g(\boldsymbol{x}_N))$, $\boldsymbol{\mu} = (m(\boldsymbol{x_1}), ..., m(\boldsymbol{x_N}))$ and $\boldsymbol{K}_{ij} = k(\boldsymbol{x_i}, \boldsymbol{x_j})$. m is the mean function and k is a positive definite kernel (covariance) function. In other words, Equation (2.3) assigns the probability for all the functions satisfying the mean and covariance conditions. After some observations $\boldsymbol{y}$ are made, where each element y_i of $\boldsymbol{y}$ is the random observation of $g(\boldsymbol{x_i})$ for $\boldsymbol{x}_i \in \boldsymbol{X}$, prior can be transformed to posterior distribution $p(\boldsymbol{g}|\boldsymbol{X}, \boldsymbol{y})$. With given posterior distribution, objective values in new

unobserved inputs $\boldsymbol{X_*}$ can then be inferred by predictive distribution(Gaussian with mean $\boldsymbol{\mu}_*$ and $\boldsymbol{K}_*$).

$$p(\boldsymbol{g}_*|\boldsymbol{X_*}, \boldsymbol{X}, \boldsymbol{y}) = \int p(\boldsymbol{g}_*|\boldsymbol{X}_*, \boldsymbol{g})p(\boldsymbol{g}|\boldsymbol{X}, \boldsymbol{y})dg = \mathcal{N}(\boldsymbol{g}_*, \boldsymbol{\mu}_*, \boldsymbol{K}_*) \tag{2.4}$$

For the equation, we denote values predicted for the set of input points $\boldsymbol{X_*}$ as $\boldsymbol{g_*}$.

Acquisition function

Acquisition function is used to select the most promising sampling point for the next evaluation of the objective function. Formally, computing the next sample point can be prescribed as optimization of acquisition function

$$x_t = argmax_x u(x|D_{1:t-1}) \tag{2.5}$$

where u is the acquisition function and $D_{1:t-1}$ are $t-1$ samples $\{\{\boldsymbol{x}_i, y_i\}|i \in \{1, ..., t-1\}\}$ observed from f.

Many acquisition functions were studied for the use in Bayesian optimization. Some of the well known are maximum probability of improvement (MPI), expected improvement (EI), or upper confidence bound (UCB).

Maximum probability of improvement function chooses point that is most likely to improve upon the minimal value of f observed so far. Since the function focuses only to the points with highest probability of improvement, unknown areas of the input space are left unexplored, which may sometimes lead to convergence to local optima. Expected improvement and upper confidence bound functions resolve this by introducing two terms to manipulate the trade off between exploitation and exploration. This makes them more robust against being stuck in local optima, allowing broader exploration of the input space. In this work, we select expected improvement as a promising acquisition function above the upper confidence bound method, since it was empirically observed to perform better than upper confidence bound in some minimization problems [2]. Also, unlike the upper confidence bound method, it does not require the setting of its own hyperparameter. As mentioned above, EI acquisition function uses exploitation and exploration parameters to trade off between the sample locations with higher probability of improvement and locations for which no observations were made yet. Suppose f' is the highest value of the objective f observed so far. The expected improvement can be defined as

$$EI(x) = \mathbb{E}[max(0, f' - f(x))|] \tag{2.6}$$

Given posterior predictive with mean $\mu(x)$ and standard deviation $\sigma(x)$, EI can be evaluated analytically [1]

$$EI(x) = \begin{cases} (\mu(x) - f' - \xi)\Phi(Z) + \sigma(x)\phi(Z), & for\ \sigma(x) > 0 \\ 0, & for\ \sigma(x) = 0 \end{cases} \tag{2.7}$$

given

$$Z = \begin{cases} \frac{\mu(x) - f' - \xi}{\sigma(x)}, & for\ \sigma(x) > 0 \\ 0, & for\ \sigma(x) = 0 \end{cases} \tag{2.8}$$

where Φ is Cumulative distribution function (CDF), ϕ is Probability density function (PDF) of the standard normal distribution, and ξ is the parameter determining the amount of exploration for the acquisition function. Higher values of ξ lead to higher amount of exploration.

For our case, we expect the observations made from the objective function to be noisy. We assume observations to be of form $\{x_n, y_n\}$, where $y_n \sim \mathcal{N}(f(x_n), v)$ and v is the variance of the noise contained in the observations. For the case of noisy observations, Noisy Expected Improvement (NEI) acquisition function was introduced in [5]. NEI is the extension of EI that allows the acquisition function to be optimized even with noisy observations. The core idea lays in replacing the value of $f^{'}$ (because we no longer know it when noise is contained in observations) with the Gaussian Process mean estimate of the best function value $g^{'} = min_{\boldsymbol{x}} \mu_f(\boldsymbol{x})$. This approach was introduced by Picheny et al. [6] as a "plug-in" strategy. As it is more suitable for our case, we choose NEI over EI to be used as an acquisition function for our implemenation.
Fitting the GP model to the observed values and optimizing NEI as acquisition function in one iteration of Bayesian optimization is shown in Figure 2.1.

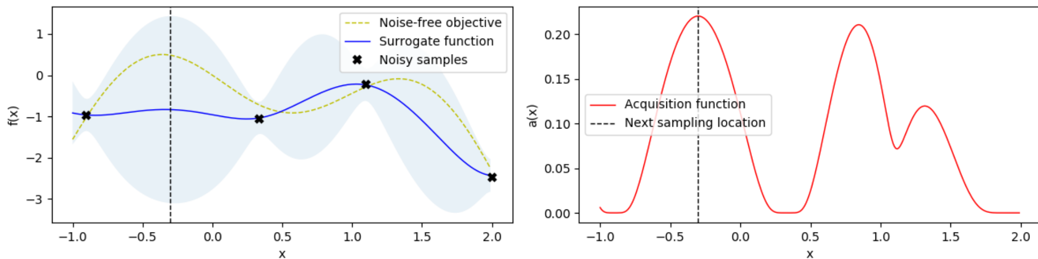


Figure 2.1: Surrogate model (on the left) is built upon the observed noisy values and is used to optimize acquisition function (on the right) and select the most promising sample for the next evaluation. Vertical dotted line shows the next sample chosen by the optimization in the given iteration.

2.2.3 Optimization algorithm

Assume we have "black-box" function f and budget of T function evaluations we can run. Bayesian optimization cycle is shown in Algorithm 1

Algorithm 1: Bayesian optimization

```
1 Define GP prior on f
2 t ← 1
3 while t <= T do
4     x_t = argmax_x u(x|D_{1:t-1})          // Find sample for the next evaluation
5     y_t = f(x_t) + v     // Observe function value with noise v in sample location
6     D_{1:t} = {D_{1:t-1}, (x_t, y_t)}         // Update the database of collected samples
7     Update the surrogate model
8 return point observed with the largest f(x) + v
```

At the beginning of the optimization, surrogate model is initialized in form of prior for the objective f (Subsection 2.2.2). On line 4, the acquisition function is called over the surrogate model aiming to find the most promising sample point for the evaluation of objective f. Lines 5 and 6 correspond to the evaluation of f and updating the set of observations with a new objective value with possible noise v returned from f. Once the observation is made, surrogate model is updated and prior is converted to the posterior over functions (line 7). By observing more values, the surrogate model is more accurate and the more information about the function can be exploited for the acquisition function in every other iteration. This factor is what makes Bayesian optimization so powerful in terms of global optimization.

2.2.4 Surrogate initialization

To avoid the need for defining the prior belief for the surrogate model in the beginning of the optimization, it can be reasonable (especially for the functions with higher dimensionality of input $x \epsilon \mathbb{R}^d$) to run the black-box function with few initial sample inputs first. These sample inputs are drawn randomly from the search space and serves as the initial observations for surrogate model to build a posterior distribution over the functions.

Out from many approaches (see [7] for overview) we selected quasi-random sobol sequence [8] as it generates random points with low-discrepancy, providing evenly spaced samples for initialization of the surrogate model.

Quasi-random Sobol sequence

Sobol sequence is a sequence, which uses base of 2 and bitwise exclusive-or operator to sample points in highly uniform manner. Given the set of d-dimensional points $\boldsymbol{p}$, where each point $p_i \epsilon \boldsymbol{p}$ is the point in the sobol sequence, the coordinte p_{ij} can be computed as

$$p_{ij} = \begin{cases} 0, & for\ i = 1 \\ \gamma_i(1)v_j(1) \oplus \gamma(2)v_j(2) \oplus, ... & for\ i > 1 \end{cases} \tag{2.9}$$

where $\gamma_i(n)$ are the binary digits of the value $i-1$, and $v_i(n)$ are uniquely defined values called direction numbers. For extensive description of the sequence and generation of direction numbers, we refer to [8].

2.3 Other methods

Other methods for hyperparameter optimization used by practicioners are grid search and random search [9]. These techniques, widely used for their simplicity and readability, have proven to be sufficient in many applications of hyperparameter optimization.

2.3.1 Grid search

Grid search is a simple procedure, which iteratively evaluates and stores the objective function for every possible parameter combination from the search space. Once all parameter configurations are evaluated, the model with the best performance observed is chosen to be optimal. Method has main the disadvantage in its computational complexity and lack of any guidance for generation of the inputs, but can be sufficient for models, where the number of function evaluations is not an issue, or the search space for parameters is small enough.

2.3.2 Random search

Random search is a method similar to grid search, extending the search by introducing randomness in the generation of the parameter inputs. Random search method chooses a predefined number of randomly drawn combinations and evaluates the objective function. Thanks to the randomly selected samples, chances of finding optimal solution in a limited amount of trials are comparatively higher than for grid search. This fact was empirically proven in [9] by comparing grid search and random search applications

for many machine learning models. Illustrative comparison between the grid search and random search procedures for two hypothetical hyperparameters can be seen in Figure 2.2.

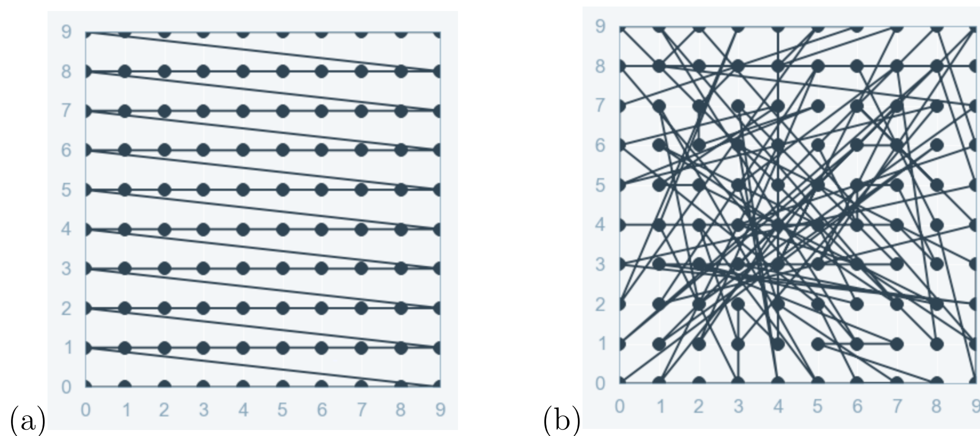


Figure 2.2: For each subfigure, two axes display the values of two illustrative hyperparameters being optimized. Points refer to individual combinations of parameters selected by the optimization method. Edges between points show the process of how individual methods, grid search (a) and random search (b), search through the parameter space.

2.4 Comparison

While grid search and random search are widely used techniques for HPO, they are not suited for the optimization of higher dimensional parameter spaces, especially in cases of expensive objective functions. From all methods described in this Chapter, Bayesian optimization seems like a best fit for our domain, as it has proven success in many applications of global optimization for machine learning models, and is well suited for optimization of the black-box function. Also, the flexibility in using different surrogate models and acquisition functions makes Bayesian optimization ideal for research and exploration of the new approaches. These properties make Bayesian optimization suitable candidate as an global optimization tool for our domain.

Chapter 3

Bayesian Optimization and Local Search Algorithms

In this chapter, we experiment with the Bayesian Optimization method and study its applicability for the hyperparameter optimization of local search algorithms. In section 3.1, some general terms that will be used furthemore in our work, are defined. In section 3.2, we select the representative set of local search algorithms that will be used in our experiments and choose the problem domain for evaluation of the algorithms. Section 3.3 validates the suitability of the Bayesian black-box optimization approach applied to local search algorithms. Finally, in section 3.4, we present a possible adaptation of the optimization approach for our domain.

3.1 Terms

To set the unified terminology for our domain, we present some general terms that will be used furthermore in our work.

Solver
Representation of the black-box function. In our case, solver is the local search algorithm for which the hyperparameter optimization is run.

Problem instance
Definition of the particular problem to be solved by the solver.

Iteration
As all of the local search algorithms used as a solver in our experiments are iterative, we define iteration as a metric for measuring the time spent by the algorithm to find solution for the specified problem instance.

Optimization trial
Optimization trial represents one execution of the black-box function (solver) done by optimization method. Input of the trial is hyperparameter configuration generated by the method for the execution. Output is the cost retrieved from the solver at the end of the execution. Maximum number of iterations the solver can take in one optimization trial is specified by the solver budget.

Solver budget
Budget of iterations available for each execution of the solver. Budget defines how many iterations can solver use in one execution during the optimization trial.

Optimization process (optimization)
Sequence of optimization trials managed by the given optimization method. Result of the optimization process is hyperparameter configuration with the best cost retrieved during the optimization trials.

Optimization budget
Budget of iterations available for the optimization process. Optimization method uses the budget by calling optimization trials repeatedly with aim to find the best hyperparameter configuration for the given solver.

3.2 Selection of local search algorithms

In the following subsections, we are going to present the local search algorithms that we selected to represent the solver for our research and show corresponding hyperparameters for each of them. We also define a specific problem instances, which will be used for the algorithms evaluation throughout our work.
Since local search algorithms are going to be treated as a black-box function for the optimization method, we do not need any knowledge about the individual hyperparameters. Therefore, range of values and type of the parameter is the only information that needs to be specified for the optimization (as shown in tables 3.1, 3.2 and 3.3). Ranges for all parameters were selected according to the documentation resources of individual local search algorithms.

3.2.1 TASP

TASP (Task and Asset Scheduling Platform) [10] is the framework developed by Blindspot Solutions [11], designed to solve NP-complete scheduling problems. Hy-

perparameters for TASP are shown in Table 3.1.

parameter	values	type
cooling	[5.0, 30.0]	continuous
initial temperature ratio	[0.1, 0.5]	continuous
mean amount to remove	[1, 50]	discrete
min amount to remove	[1, 15]	discrete
max amount to remove	[15, 70]	discrete
deviation amount to remove	[0.0, 1.5]	continuous
dead end	[0.0, 1.0]	continuous
update interval	[70, 80, 90, 100, 110, 120]	discrete
best solution reward	[7.0, 13.0]	continuous
accepted solution reward	[0.1, 2.0]	continuous
rejected solution reward	[-2.0, -0.1]	continuous
relative minimum weight	[0.0, 1.0]	continuous
relative maximum weight	[1.0, 5.0]	continuous
decay factor	[0.0, 1.0]	continuous

Table 3.1: Hyperparameters definition for TASP

3.2.2 Jsprit

Jsprit is open source tool [12] for solving rich Traveling Salesman problems (TSP) and Vehicle Routing problems (VRP), based on a single all-purpose meta-heuristic. Hyperparameters that can be configured for the algorithm are prescribed in Table 3.2.

3.2.3 OptaPlanner

Opta planner is another open source local search solver [13], capable of optimizing complex planning and scheduling problems such as TSP, VRP, Task Assignment, School Timetabling and many others. Due to the limited documentation resources for the algorithm, we were not able to identify more than 4 hyperparameters for the solver configuration. However, we consider it as an opportunity to compare the performance of resulting optimization method on hyperparameter sets of different dimensions.

parameter	values	type
radial best	[0.0, 1.0]	continuous
radial regret	[0.0, 1.0]	continuous
random best	[0.0, 1.0]	continuous
random regret	[0.0, 1.0]	continuous
string best	[0.0, 1.0]	continuous
string regret	[0.0, 1.0]	continuous
k-min	[0, 5]	discrete
k-max	[6, 10]	discrete
l-min	[5, 10]	discrete
l-max	[11, 40]	discrete
worst best	[0.0, 1.0]	continuous
worst regret	[0.0, 1.0]	continuous
cluster best	[0.0, 1.0]	continuous
cluster regret	[0.0, 1.0]	continuous

Table 3.2: Hyperparameters definition for Jsprit

parameter	values	type
entity tabu ratio	[0.01, 0.99]	continuous
fading entity tabu ratio	[0.01, 0.99]	continuous
water level increment ratio	[0.01, 0.99]	continuous
value tabu ratio	[0.01, 0.99]	continuous

Table 3.3: Hyperparameters definition for OptaPlanner

3.2.4 VRPTW

For the evaluation of the selected solvers, some benchmark problem must be specified. We choose Gehring & Homberger's [14] VRPTW (Vehicle Routing Problem with Time Windows) benchmark instances for 1000 customers as an input problem for the evaluation. VRPTW is the extension of classic VRP (Vehicle Routing Problem), for which the set of vehicles is defined to operate the set of customers. In VRPTW, every customer has additionaly the time interval assigned, in which he can be served.
As selected instances are standard benchmark instances for combinatorial optimization, we assume that they are complex enough for the algorithms to represent the long-running black-box objective function. However, any other benchmark problem

compatible with the selected algorithms and complex enough to fulfill the ”expensiveness” requirement would suffice. Set of problem instances used for our experiments is prescribed in Table 3.4.

Table 3.4: Selected set of problem instances

Name	C110_1	C110_2	C110_3	C110_4	C110_5	C110_6

Every instance contains problem definition for 250 vehicles and 1000 customers.

3.3 Bayesian optimization analysis

In this section we aim to validate the usefulness of the Bayesian optimization method for the HPO of the solvers presented in Section 3.2. We implement a prototype of classic Bayesian optimization method and run experiments on HPO comparing Bayesian optimization with random search as a competitive candidate method mentioned in Chapter 2.

3.3.1 Implementation

For our experiments, we implemented a prototype of Bayesian Optimization method. For the purpose of our research, we used Adaptive Experimentation Platform (Ax) [15] released by Facebook Inc. in 2019. Platform is using BoTorch [16, 17] library for Bayesian Optimization, providing all the management functions around BoTorch. Ax allows extensive configuration of the optimization process along with the tools for evaluating the experiment results, which makes it a great tool for our analysis.

Bayesian optimization has many possibilities of how to set the surrogate model and acquisition function for the method. As we expect the observations from the solvers to be noisy[1] we use NEI (Subsection 2.2.2) as an acquisition function in our implementation. For Surrogate model, we choose GP (Subsection 2.2.2) as it is a model widely used in Bayesian optimization method and can serve as a convenient baseline for the future experiments.

To compare performance of Bayesian optimization, we also implemented prototype of random search optimization method.

[1]Algorithms are not deterministic, returning different objective values (costs) for the same configuration when evaluated multiple times

3.3.2 Approach verification

Given our prototype of selected optimization methods, we performed HPO experiments for each solver in order to process and compare the results of each method with each of the solvers optimized as a black-box.
Our experiment is defined as a set of independent hyperparameter optimization processes of the given black-box function (solver), using one of the selected HPO methods. Each optimization process in the experiment runs for the specified number of optimization trials[2], searching for the best combination of hyperparameters. Performance of each optimization process is recorded and, once sufficient amount of optimization processes is done, results are aggregated to display the statistical performance of the given method for the given solver.
As our goal is to compare the performance of individual optimization methods, the setting used to run each of the solvers was fixed as described in Table 3.5.

Solver	Solver budget	Problem instance
TASP	10,000	C110_1
JSprit	500	
OptaPlanner	100,000	

Table 3.5: Solver budget column defines the number of iterations, for which the solver runs as a one black-box function call. Problem instance contains definition of the problem, that is being solved by the given solver.

As we see in the table, the budget of iterations specified for each solver differs in size. Since every solver defines iterations differently, we chose the values empirically by running individual algorithms, measuring its runtime and solution costs, identifying appropriate limits for the purpose of our experiments.
To validate Bayesian Optimization and compare it with random search method, we performed experiments with following properties

- Every experiment consists of 10 optimization processes
- Every optimization process in the experiment is ran for 30 optimization trials

The results of experiments can be seen in Figure 3.1. For each experiment, given all optimization processes performed during the experiment, median values of cumulative minimums of all optimizations are displayed for each optimization trial[3]. The median

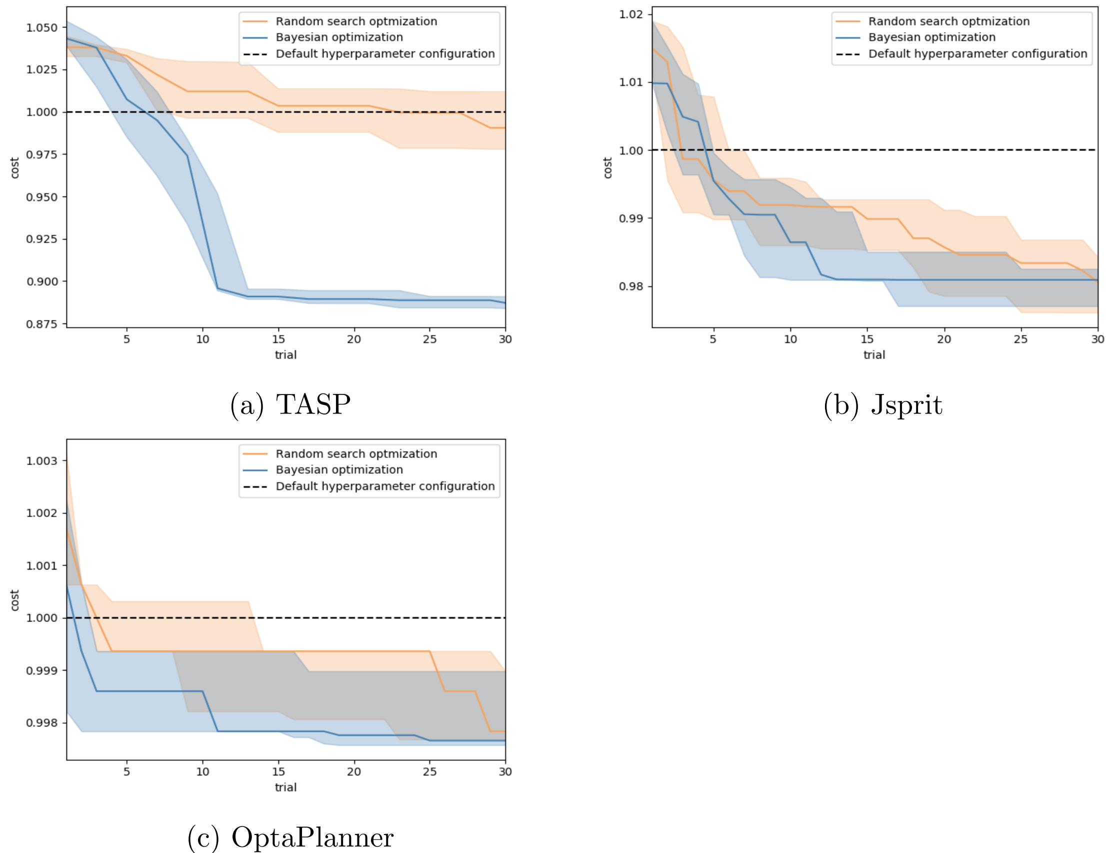
[2] One configuration of hyperparameters is evaluated in each trial

[3] Cumulative minimum for trial t is minimum of all values obtained from the beginning of the optimization until the respective trial t

value was chosen as an optimal statistical property due to the noisiness in the outputs of the selected solvers. To illustrate the distribution of optimization trials around the median, area between 1st and 3rd quartile is shown for each method. Also, to highlight the outcome of each optimization method, results are normalized as

$$c_* = \frac{c}{d} \tag{3.1}$$

where c is the original objective cost returned by the solver for the given hyperparameter setting on the fixed problem instance and d is the objective cost retrieved by evaluating respective solver in default hyperparameter setting (without optimization). Therefore, for $c_* > 1$ there is a higher cost found by the solver for the given hyperparameter setting compared to the default hyperparameters in selected hyperparameter configuration, whereas for $c_* < 1$ there is an improvement achieved by optimization method.



(a) TASP

(b) Jsprit

(c) OptaPlanner

Figure 3.1: Each subfigure displays comparison between Bayesian optimization (blue) and Random search (orange). For each optimization trial, median of cumulative minimums of normalized costs from each optimization is shown. Horizontal dashed line represents the cost observed for the default configuration of hyperparameters (without any optimization).

The biggest difference between optimization methods was observed for TASP solver,

where the Bayesian optimization outperformed Random search significantly. For other solvers, the difference in results is less notable. Nevertheless, the Bayesian optimization converged faster for both Jsprit and OptaPlanner. From the Figure 3.1, we can also deduce the dependence of individual algorithms on their hyperparameter configuration. For TASP, there is a clear dependence between chosen hyperparameters and performance of the algorithm. For Jsprit, we see that after 30 configurations chosen by the optimization method, performance did not improve much. This tells us that algorithm is less prone to changes in hyperparameter configuration. For such case, optimization method may need to run longer to achieve the significantly higher improvement. Opta planner also did not show higher performance improvement, which could be caused by low dimensionality of its hyperparameters.
According to the results obtained from our experiments, we consider Bayesian optimization as a promising tool for a black-box optimization of the local-search algorithms.

3.4 Domain adaptation

General approach of the Bayesian optimization method is limited by the fact, that objective function is treated as a black-box. This factor completely decouples the Bayesian approach from the properties of the objective function, as only information provided by the function is the objective value. Since, in our case, the domain of the objective function is known to us, we are not limited by the assumption of the black-box. Therefore, we see the possibility of improvement when the domain is revealed for the optimization method. In this section, we discuss the possible extension of the classic Bayesian approach and analyse the properties of the selected solvers that could be used for performance improvement.

3.4.1 Solver progress curve

Typically, progress of the local search algorithms can be measured by iterations. For every iteration, the algorithm tries to find an improvement in the solution for the given problem. Once the limit of iterations (solver budget) specified for the solver is exceeded or another termination condition is satisfied, algorithm stops. In every iteration, the cost of the best found solution can either stay constant or decrease (while improvement is found by the algorithm). Given this, we can characterize the progress of the solvers as a monotonic time-series, where the time unit is represented by iteration and value is the cost of the best solution found so far. Furthermore, we refer to this process as to "solver progress curve".

To get the general overview on the behavior of the solvers, we performed analysis of the progress for each of the selected solvers. We ran each solver multiple times for different, randomly generated hyperparameter configurations to see if there are any trends in the progress, independent of the selection of the hyperparameters. Individual experiments displayed as a time-series are shown in Figure 3.2. All cost values in the Figure are normalized as

$$c_* = \frac{c}{c_{min}} \tag{3.2}$$

where c is the cost value and c_{min} is the minimum of all cost values observed in the experiment.

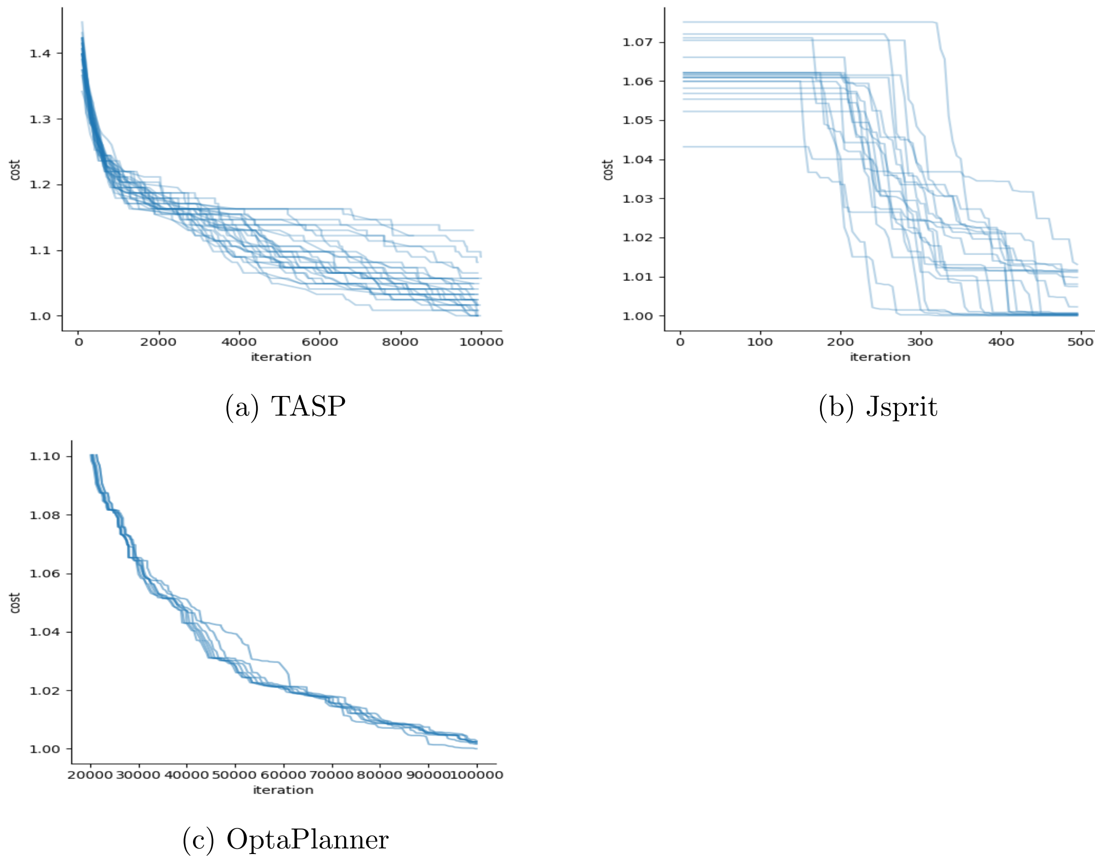


Figure 3.2: For each subfigure, each plot line represents one run of the given local search algorithm, where for every iteration point (x-axis) there is a normalized cost of the found solution displayed (y-axis).

From the Figure, we see that the trend is most visible for OptaPlanner, where there is obvious correlation for all the runs. The worst descriptive is the progress of JSprit with the constant values until around 200 iteration. Nevertheless, with TASP solver in the middle, we see that, despite the different hyperparameter setting, specific stochastic trend in a progress curves is visible for each of the solvers.

3.4.2 Early stopping

According to the performed experiments, we consider analysed time-series data as a useful property for extension of Bayesian optimization. Given visible trend in the time-series, hyperparameter tunning could be accelerated by predicting the future time series data of the individual objective function (solver) evaluations and discarding those, for which prediction is not promising. When prediction is sufficiently fast, this modification could result in higher number of hyperparameter configurations that can be evaluated for the optimization and so the chances of finding optimal configuration would increase. The idea stands on the assumption, that time-series data obtained from the solver execution, are predictable. In the following subsections, we study techniques that could be used for extrapolation of the time-series derived from the solver progress curve, and validate their usefulness for our case. We focus on ARIMA model and Random Forest Regression as a promising tools for such task.
To specify the time units for the time-series of the selected solvers, we present another term, that will be used in the following sections.

Iteration split factor

Defines the number of iterations representing one time unit of the time series data derived from the solver progress curve. The term is used to unify the definition of time units between different solvers.

To test the performance of the studied techniques, we created time-series dataset for each of the solvers. To create each dataset, algorithm was executed multiple times for the given iterations (fixed setting is prescribed in Table 3.5) and the solution cost values were recorded according to the iteration split factor configuration[4]. Illustrative example of a dataset is shown in Table 3.6.

Table 3.6: Example of time-series data

Iteration	100	200	300	400	500	600	700	800	900	1000
Cost values	1.6	1.5	1.4	1.3	1.2	1.0	0.98	0.95	0.92	0.9

For each algorithm, we gathered 20 time-series (using randomly generated hyperparameter configuration for each run), each with 100 time-units. These datasets are used furthermore to test and validate the performance of the individual prediction techniques.

[4]Assuming the factor value is k, cost values are stored for every k iterations of the solver

3.4.3 ARIMA model

Auto Regressive Integrated Moving Average (ARIMA) is a model used in analysis and forecast of non-seasonal time-series[5]. Model was applied in many fields [18, 19] and is well known tool for prediction of those time-series data which exhibits patterns.
To dive deeper into the problematic, we mention several terms to understand how the ARIMA model manipulates with time-series data.

Lag
Lag is a time (number of time units) between two time series.

Autocorrelation
Autocorrelation is a measurement of how the time-series is correlated with its past values. Typically, autocorrelation is being displayed as autocorrelation function plot, where the correlation coefficient is on the x-axis, and y-axis refers to the number of lags.

Stationarity
Stationarity is a property of time-series, which indicates that the mean and variance of the data are constant over time.

Differencing
A method to transform non-stationary data into stationary one, done by calculating differences between time-series and its lagged version. Differencing can be applied multiple times to make given time-series stationary.

ARIMA model is defined by three variables, that predetermine how the time-series must be modified in order to make prediction possible, and specifies the way, how the prediction is calculated.

I (Integration term)

Integration term refers to order of differencing needed to make given time-series stationary. Stationary data are required for the model.

P (Autoregressive term)

Autoregressive term is order of the Auto Regressive (AR) term. Variable defines number of lags that will be used as predictors for the predicted value. Using autoregressive

[5] Time-series without presence of regularly repeating variations

term, function for the prediction is

$$Y_t = \alpha + \beta_1 Y_{t-1} + \beta_2 Y_{t-2} + .. + \beta_p Y_{t-p} \tag{3.3}$$

where $Y - t$ is the predicted value, Y_{t-1} is the lag of the time-series, β is the coefficient of corresponding lag that is estimated by the model and α is intercept term also estimated by the model.

Q (Moving average term)

Moving average term defines a lag of the error component for the calculation. Error component refers to data in time-series, which does not follow trend or seasonality. Prediction for the model using only moving average term is

$$Y_t = \alpha + \phi_1 \epsilon_{t-1} + \phi_2 \epsilon_{t-2} + .. + \phi_q \epsilon_{t-q} \tag{3.4}$$

where Y_t is the predicted value, α intercept term estimated by the model as well as ϕ coefficients. ϵ_{t-1} is error of the autoregressive model of lag 1.

ARIMA model is prescribed as a model, for which at least one order of differencing was used and P and Q variables are set to non-zero values. Then, the predicted value is calculated as

$$Y_t = \alpha + \beta_1 Y_{t-1} + \beta_2 Y_{t-2} + .. + \beta_p Y_{t-p} + \phi_1 \epsilon_{t-1} + \phi_2 \epsilon_{t-2} + .. + \phi_q \epsilon_{t-q} \tag{3.5}$$

To build correct ARIMA model, these variables must be set according to the properties of the data. The process of setting a variables typically includes extensive analysis of the data and is crucial for the performance of the model.

Prototype

To validate ARIMA technique on our time-series datasets, we implemented prototype that is able to predict the future value of the given time-series at desired time step $t + w$, where t is the last observed time unit of the time-series and w is the time window for which we want to predict the value. Prototype uses Equation 3.5 to predict the consecutive values of the time-series until the desired time unit $t + w$ is reached. Given the prototype, we tunned the I, P and Q variables on the datasets to choose the most convenient configuration for the model. Best performance was observed for the

following configuration.

$$I \leftarrow 1, P \leftarrow 1, Q \leftarrow 1 \tag{3.6}$$

Despite the best performing configuration, the model performed poorly for all of the time-series. Two examples of ARIMA prediction applied to time-series gathered from the TASP algorithm are shown in Figure 3.3.

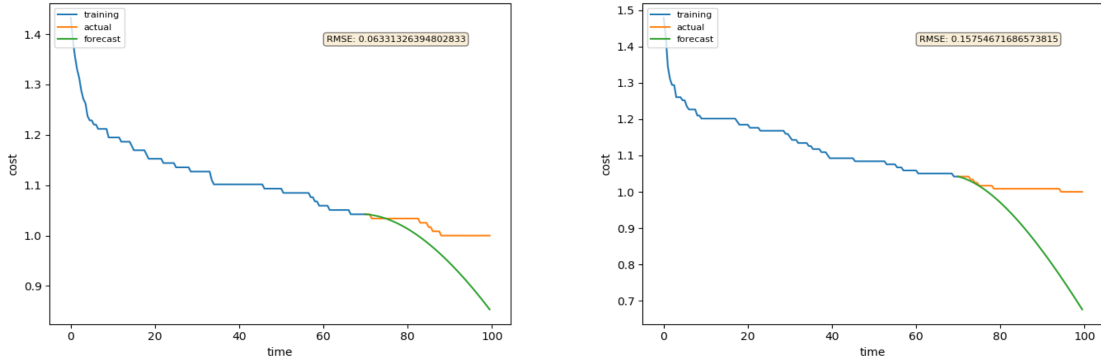


Figure 3.3: Cost values are displayed on y-axis for a given time unit (x-axis) of the dataset. Blue line refers to data, from which the model is trained to make a prediction of future values (green line). Orange line shows the real cost values that were observed for the particular time-series.

Due to our results on ARIMA model, we do not consider the technique suitable for our case. As it turned out, the core problem in ARIMA application on our data lies in inconsistency of properties of individual time-series like stationarity and autocorrelation. Since, order of differencing needed to make data stationary can differ between individual time-series, model would need to reinitialize multiple times when applied to the bigger amount of data. Also, the model accuracy decreases distinctly as a window for prediction increases, which makes the model hardly usable for our case.

3.4.4 Random Forest regression

Random Forest [20] is a machine learning algorithm, that combines multiple decision trees together and aggregates them into an ensemble. Ensemble method makes Random Forest less vulnerable to overfitting or being stuck in a local optima, which makes it outperform any individual decision tree model. Model is suitable for both classification and regression. For our case, we analyse the use of Random Forest regression for prediction of the time-series datasets.

Random forest model was studied as a promising tool for the time-series forecasting [21] and already showed success in several applications in a field, e.g., stock index movement forecast [22] where it outperformed neural networks and traditional discriminant and

logit models used for time-series prediction.
As Random Forest is, unlike ARIMA model, a general machine learning method, it requires the feature extraction from the data as an input for training and prediction of the model. Selection of the features is crucial for the performance of the model and must be chosen respectively to the properties of the given time-series. Appropriate features suitable for the time-series prediction were studied in [22, 23].

Features

From the study mentioned above, we selected 4 feature variables as a candidate features for the Random Forest time series prediction. Features were chosen with emphasis on balance between their computational complexity and information gain.
For the following equations, we define C_t as a cost value C observed at given step t and n as a time span[6] for which we calculate the feature value.

Simple Moving Average

Simple Moving Average (SMA) is typical feature used for time series prediction. It is unweighted mean of costs in previous n iterations.

$$SMA(n) = \frac{1}{n}\sum_{i=0}^{n-1} C_{t-i} \tag{3.7}$$

Momentum

Momentum measures the amount of change in cost over a given time span.

$$M(n) = C_t - C_{t-n} \tag{3.8}$$

ROC

ROC can be understood as rate-of-change. It describes the difference in current cost and the cost n steps ago.

$$ROC(n) = \frac{C_t}{C_{t-n}} * 100 \tag{3.9}$$

Disparity

Distance of current cost and the simple moving average of n steps.

$$DP(n) = \frac{C_t}{SMA(n)} * 100 \tag{3.10}$$

[6] Number of previous steps for which the feature variable is calculated

According to the length of solver progress curve available to the Random Forest model, feature variables need to be chosen reasonably and their time spans cannot exceed the number of steps available for the training of the model.

Regression task

In order to specify the regression task for the regressor, we consider the solver progress curve as a monotonic series. Due to the nature of local search algorithms, we know that, with increasing iteration number, the cost is always decreasing with new best solutions found during the process.
Formally, we define the predicted cost for step t as

$$C_t^* = C_t - C_{t-1} \tag{3.11}$$

where C is the cost value and t is the time step for which the prediction is called. Due to the monotonicity of the solver improvement curve, the prediction will always be trained with the positive values for the regression[7]. Therefore, the monotonic constraint is ensured even for the predictions done on unknown data. Since we want the prediction values to follow the monotonicity of the original improvement curve, this is desired property for our case.

Prototype

For the purpose of our research, we performed experimental analysis to identify the best subset of features suitable for our case. We implemented a prototype of Random Forest regression using sklearn [24] implementation of Random Forest Regressor. The prototype was then trained and tested on 50 different variations of selected feature variables extracted from our datasets, measuring the performance with RMSE metric. To test the performance of the model, 5-fold cross-validation technique was used. According to the analysis, combination of three particular features showed the best performance for the model.

- Simple Moving Average (SMA)
- Momentum (MO)
- Rate of Change (ROC)

Combination of features that showed the best results can be seen in Table 3.7, where column n refers to the period of lags for which the feature is calculated.

[7] Representing the decrease in solution cost

Table 3.7: Feature set for Random Forest regression

Feature	**n**
SMA	50
SMA	30
SMA	10
MO	5
MO	2
MO	1
ROC	10
ROC	5
ROC	2

For the given feature set, we performed the prediction experiments for all datasets as can be seen in Figures 3.4, 3.5 and 3.6.

Figure 3.4: Random Forest regression for TASP time-series data

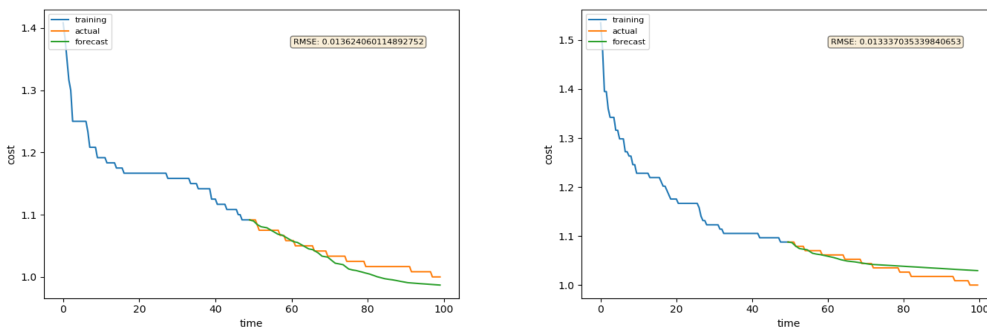


Figure 3.5: Random Forest regression for Jsprit time-series data

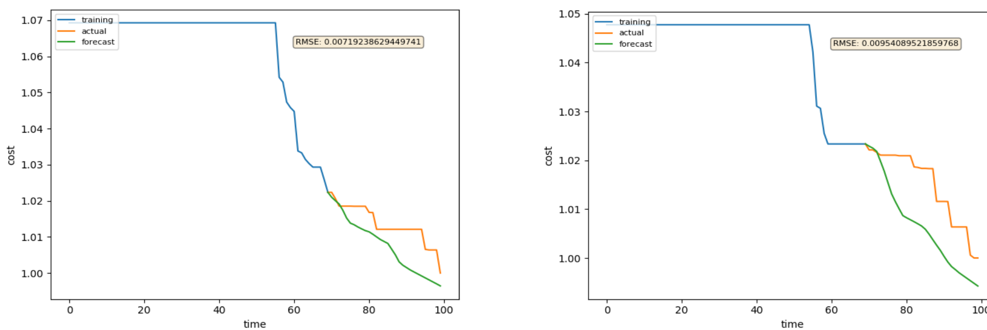
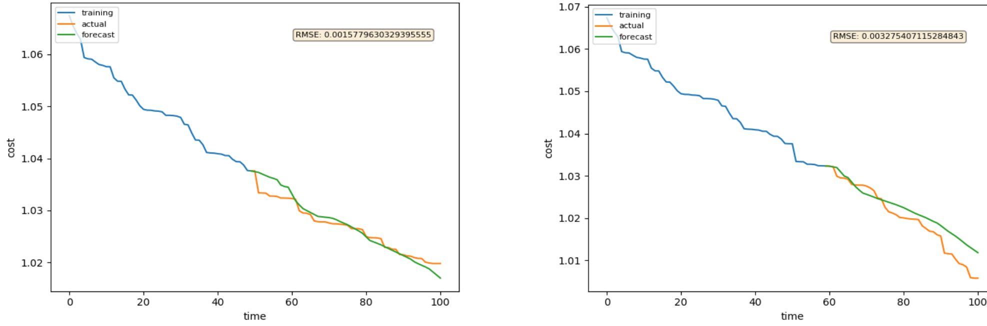


Figure 3.6: Random Forest regression for OptaPlanner time-series data



As seen in the Figures, best quality of the prediction seems to be observed for TASP, with the prediction curve copying the actual values. Worst performance was noted for Jsprit, for which the first half of the time-series is unusable due to the underlying search strategy of the algorithm. However, the metrics obtained for Jsprit are still significantly better than for ARIMA model.

3.4.5 Comparison

According to our research, Random Forest showed significantly better results for our datasets than ARIMA model. Comparison of the RMSE metrics for both models is shown in Table 3.8.

	Random Forest	ARIMA
TASP	**0.0188**	0.2905
Jsprit	**0.0623**	0.2911
OptaPlanner	**0.0407**	0.1727

Table 3.8: Mean of RMSE metrics measured for all test time-series data for both Random Forest and ARIMA model. Minimal values are highlighted for each solver. To make the RMSE metrics comparable, original values and the prediction values were normalized to range $[0, 1]$ before computing the metric.

.

We see main advantage of the Random Forest regression in its ability to learn from the multiple time-series. This fact makes the method more accurate over time, when new observations are made and model is trained with more data (unlike ARIMA model, where prediction is always based on the properties of individual time-series). Also, it can be deduced from the experiments, that, when correct set of features is specified, Random Forest requires less configuration to work properly on a different time-series data. This is what makes it more robust than ARIMA model, especially when the long

time prediction (explained further in Section 4.3), is desired. Due to all mentioned properties of the Random Forest regression model, we consider it as a promising tool for prediction of the gathered time-series data.

Chapter 4

Algorithms

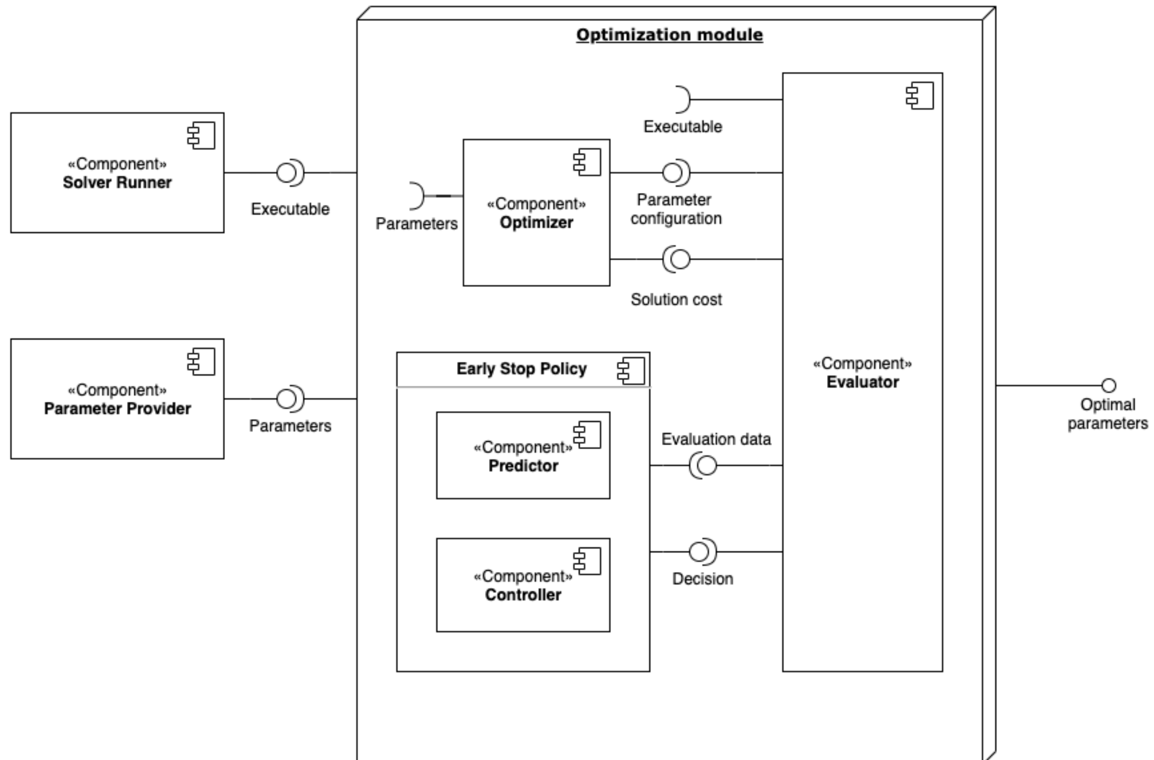
In this chapter, we present our Early Stop Bayesian Optimization (ESBO) framework, which extends the classic Bayesian Optimization algorithm by introducting the Early Stop policy to control the run of the black-box function being optimized. Policy uses knowledge about the progress of the black-box function (solver) and stops the run if the progress does not seem promising for the given hyperparameter configuration.
In section 4.1, overall architecture of our optimization framework is described. Sections 4.2, 4.3, 4.4, and 4.5 are dedicated to the description of individual components and their use in the optimization framework.

4.1 Overall architecture

General purpose of the framework is to run the optimization process to optimize hyperparameter configuration for the specified solver. Every optimization process triggered by the framework has specified optimization budget, which is being spent by each optimization trial until it is depleted and the best configuration is chosen from the executed trials. The overall optimization budget needed to find high-quality hyperparameters is decreased by integrating the Early Stop policy for individual optimization trials. Policy discards the trial once its progress does not seem promising for the optimization process, leaving the remaining iterations from the solver budget unused, free to be taken by the next trials. Thanks to this selective policy, number of trials can be increased for the given optimization budget and so the number of iterations needed for the optimization process to find the better hyperparameter configuration can be decreased significantly.

ESBO framework consists of three main components, Optimizer, Evaluator and Early Stop policy. These three components create core building blocks of the framework and define the interface for the configuration and run of the optimization process. In order to start the optimization process, solver and hyperparameter space specification must be provided. Since the objective for the optimization process is treated as a black-box function, the framework provides a clear interface that defines the communication needed between the optimizer algorithm and the solver being optimized. The interface consists of two parts, solver runner, which represents the interface to run and observe the given solver implementation, and parameter provider, which defines the parameter space for the solver. Given this specification, Optimizer component is used to run the optimization process, calling the Evaluator component repeatedly to evaluate the given solver with the hyperparameter configuration chosen for the optimization trial. The overall architecture of the optimization framework is shown in Figure 4.1.

Figure 4.1: Optimization framework architecture



4.1.1 Solver specification

Solver and its parameters are specified in form of **Solver Runner** and **Parameter Provider** components. The interface for these components was implemented to make the objective definition flexible and decoupled from the optimization framework. This way, new solvers can be specified without touching the optimization framework architecture.

Solver Runner

Solver Runner implements the functionality to run and read the results of the underlying solver algorithm with given problem instance and hyperparameter configuration. This functionality is being executed and supervised by the Evaluator component, which has responsibility for evaluating the results coming from the Solver Runner component and process them for the Optimizer component[1].

Parameter Provider

Every Solver Runner needs to have corresponding Parameter Provider specified. Provider defines the hyperparameter search space for the optimization process and follows unified structure, which fully describes the properties for each parameter. Structure is shown in Table 4.1

<table>
<tr><th>Property</th><th colspan="3">Type</th></tr>
<tr><td>Name</td><td colspan="3">Varchar</td></tr>
<tr><td rowspan="2">Type</td><td colspan="3">Enum</td></tr>
<tr><td>Fixed</td><td>Choice</td><td>Range</td></tr>
<tr><td>Value</td><td>Number</td><td>List[Number]</td><td>Tuple[Number]</td></tr>
</table>

Table 4.1: Each parameter input for the optimization process consists of name, type and value properties. Name represents its unique identifier in the context of hyperparameter optimization. Type refers to the type of the parameter. There are three possible types. Fixed type defines the fixed value, choice type specifies the list of discrete values that can be tried by the Optimizer. Finally, the range type defines the range of continuous values that the parameter can take

[1]Evaluator can be understood as a middleware between the Solver Runner and Optimizer components

4.1.2 Early Stop policy overview

As part of ESBO, the Early Stop policy is used to control the execution of the underlying solver. Policy is aimed to streamline the usage of the given optimization budget by early stopping of the not promising optimization trials.
Early Stop policy consists of two main components, Predictor and Controller. Predictor is responsible for making predictions of the final objective values for the executed solver. Given data from the solver progress curve, it uses Random Forest regressor to extrapolate the curve for future iterations and predicts the cost at final iteration point. Controller component represents the behavior of the decision making policy. Given the data provided by Evaluator, Controller calls Predictor to make predictions about the curve and generates the decision about the "early stopping" of the current optimization trial. Given this, Early Stop policy reduces the budget spent by the optimization process and allows Optimizer component to perform more optimization trials in order to find the best performing hyperparameter configuration.

4.2 Evaluator

Evaluator component implements a lifecycle of what we call the optimization trial, taking responsibility for evaluating the solver and returning the cost for the hyperparameter configuration given by the Optimizer component (see Figure 4.1). The core concept of the Evaluator lays in its communication with the Early Stop policy to control the discardment of not promising executions of the solver and speed up the optimization process. To fully describe the Evaluator functionality, its key aspects are described in the following subsections.

4.2.1 Solver evaluation

As solver can be executed for different problem instances, we assume that optimal hyperparameter configuration can differ for each individual instance. Therefore, to make optimization process robust against overfitting for the specific problem instance, it is required for the Evaluator to have the ability of running solvers (through Solver Runner component) for different problem instances simultaneously, representing one execution of the objective function for optimization process. By executing the solver with multiple problem instances, optimization process is more likely to find the hyperparameter configuration optimal for broader set of problem instances, identifying the high quality optima for hyper-parameter values.

For every problem instance, the solution cost found by the solver can be of different range. As Evaluator must return one cost value as a objective value for the selected hyperparameters[2], it is necessary to aggregate the costs obtained from individual problem solutions. The aggregation must be done in such a way, that the change in objective value observed for individual problem instances is balanced. Therefore, ranges for solution costs need to be normalized. Once the normalized cost is ensured for every executed instance, aggregated objective cost can be provided by Evaluator.
For the normalization of the given problem instances, the Evaluator executes the Solver Runner at the start of the optimization process with default hyperparameters. For each instance, the execution is done with a maximum iteration budget (solver budget) and the problem-solution costs are retrieved. Once the cost retrieval process is done, the costs found for each instance are cached as a reference costs for the evaluation process. Given computed reference costs, the objective value for the optimization trial is formally defined as

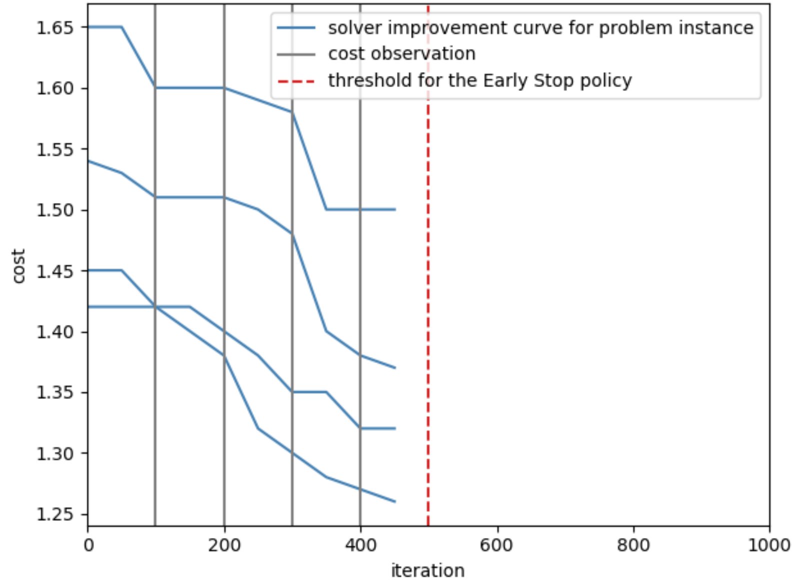
$$c = \frac{\sum_{i=1}^{n} \frac{c_i}{r_i}}{n} \tag{4.1}$$

where c_i is the cost of ith problem instance, r_i is the reference cost for the ith instance and n is the number of instances being executed.
Normalization function is defined as part of the Evaluator component and is necessary in cases, when multiple instances are evaluated during the optimization trial. To improve the performance of the normalization process, caching functionality is integrated into the Evaluator, to store the reference costs for future use of the same solver specifications. Once costs are cached, they can be called by next optimization trials without the need for executing the solver for each instance again.

4.2.2 Time series extraction

When execution of the solver is started by the Evaluator, its progress curve is transformed to the suitable time-series dataset on runtime, and provided for Early Stop policy component. Time-series are extracted using iteration split factor (described in Subsection 3.4.2) variable to define the time units for the data. Illustrative example of time-series extraction is shown in Figure 4.2.
Data observed during the solver execution serves as a source of information for the decision logic of the Early Stop policy. All data observations are also stored between the optimization trials. This makes the policy more powerful with the increasing number of optimization trials executed.

[2]This loss value is used by optimization process to guide the search for optimal hyperparameter configuration



(a)

Step	1	2	3	4	5	6	7	8	9	10
Cost values	1.53	1.52	1.5	1.4						
	1.65	1.6	1.58	1.5						
	1.45	1.4	1.32	1.28						
	1.42	1.42	1.4	1.38						

(b)

Figure 4.2: This is the hypothetical example of one optimization trial executed for 4 problem instances and solver budget of 1000 iterations with iteration split factor set to 100 iterations. That way, solver observations are split into 10 steps $(100, 200, 300, ..., 1000)$ for each problem instance. Figure (a) shows the improvement curves for each problem instance with vertical lines representing iterations at which the costs were stored for the Evaluator. Red dotted vertical line shows the step, from which the Early Stop policy is able to make decisions (Due to sufficient amount of steps stored). This threshold is defined by the configuration of the Early Stop policy component. Figure (b) shows the structure of the costs being stored. Red dotted line shows, as in the figure (a) moment of the first decision for the Early Stop policy

4.3 Predictor

Predictor component is responsible for making predictions about the solution cost using solver progress curve observations. These predictions are crucial for the decision generation of the Controller component. Given the time series data from the progress curve (As shown in Figure 4.2) of the current solver execution, Predictor performs a prediction of the solution cost at final step of the solver execution. For this type of regression problem, we use the Random Forest Regressor implementation from the scikit-learn library [24]. According to Chapter 3, Random Forest Regressor proved its usefullness in prediction of time series data extracted from the solver progress curve.
To train the regressor, solver progress curve observations from individual optimization trials are used. As the number of trials increases over time, regressor is provided with the bigger dataset to be trained on.
Feature variables used by Predictor component for training and prediction of the underlying regressor model were selected according to the research done in Section 3.4.2 (complete set of feature variables can be seen in Table 3.7)

4.3.1 Prediction with time window

Predictor component is aimed to be used for predicting the value in a given time window[3]. This type of long term forecast can also be viewed as a concatenation of multiple "next value"[4] prediction calls. Our predictor solves the challenge of a long time window intuitively by appending the predicted values to the original time-series data as a source for the next predictions. This way, the predictor is able to predict recursively until the desired time point is reached. Three main functions of the predictor are used in the prediction process.
Detailed description of the functions can be seen in Algorithm 2. *train* function expects regressor model with data as input and takes responsibility for extracting feature variables (by calling *extract* function) and training the given regressor. Once the regressor model is trained, *forecast* function can be called to predict the cost value given historical data and specified time window.

[3]Prediction of the cost value at step $t + m$ for $m \epsilon N^+$ given cost values for steps $\{1, ..., t\}$
[4]Predicton of the cost value at step $t + 1$ given known cost values for steps $\{1, ..., t\}$

Algorithm 2: Predictor component

```
   Input: Extractors E
   Data: Regressor model R
1
2  Function train(data):
3      y ← data.getLabels()
4      X ← extract(data.getValues())
5      R.train(X, y)

6
7  Function forecast(data, window):
8      H ← data                                   // Historical data
9      i ← 0                                      // Predictions made
10     while p < window do
11         X ← extract(H)
12         x ← getLast(X)                         // Feature set for last observed data
13         p ← R.predict(x)
14         H ← H ∪ p
15         i ← i + 1
16     return getLast(H)                          // Return last prediction
17
```

4.4 Controller

Controller component defines an interface for configuration and administration of the prediction process. The primary intent is to decouple prediction logic from the other components and provide them with easy to use, configurable interface. Controller takes values observed from the solver progress curve as an input from the Evaluator, and calls Predictor component to predict the cost values for the final step of the curve. With these predictions, decision is made by the Controller and sent to the Evaluator component as an output.

To configure the behavior of the Controller component, following parameters are defined.

Discardment limit

As the accuracy of the predictor component can vary according to its configuration, sometimes it can be beneficial for the controller to continue the evaluation even though the predictor marked the evaluation as unpromising. In cases, where the solver progress curve is highly unpredictable, predictions can be very noisy and contain significant error. Therefore, when discardment limit d is defined for the controller, evaluation

is discarded by the controller only if the predictor component marked evaluation as unpromising d times in a row. Discardment limit variable can be understood as a "rate of trust" in the predictor controller.

Retrain interval

Retrain interval defines the frequency for which the Predictor component is retrained with a new values obtained from the last optimization trials. The variable must be chosen reasonably according to the optimization process. High value can improve the time performance with the loss of accuracy in the beginning of the process, whether the low value increases accuracy by adapting Predictor component to the new values more often.

First prediction step

First prediction step variable defines at which step of the solver progress curve the Controller can start the prediction process. If the parameter is not provided, the value is inferred from the Predictor settings and is set to the first possible step for which the prediction can start (from the point of Predictor component).

Predictions limit

Prediction limit refers to the number of predictions the controller is able to make during one optimization trial. This variable can impact the performance of the control process significantly since it defines, how often the Predictor is called to make a prediction.

Threshold limit

To make a decision about whether to stop or continue the evaluation process, Controller calculates dynamic threshold value used to split the predictions to the promising and unpromising sets. Given the sorted set of all final cost values[5] $\{c_1, c_2, ..c_n\}$ observed in all previous optimization trials for which $c_1 \leq c_2 \leq .. \leq c_n$, the threshold value is defined as

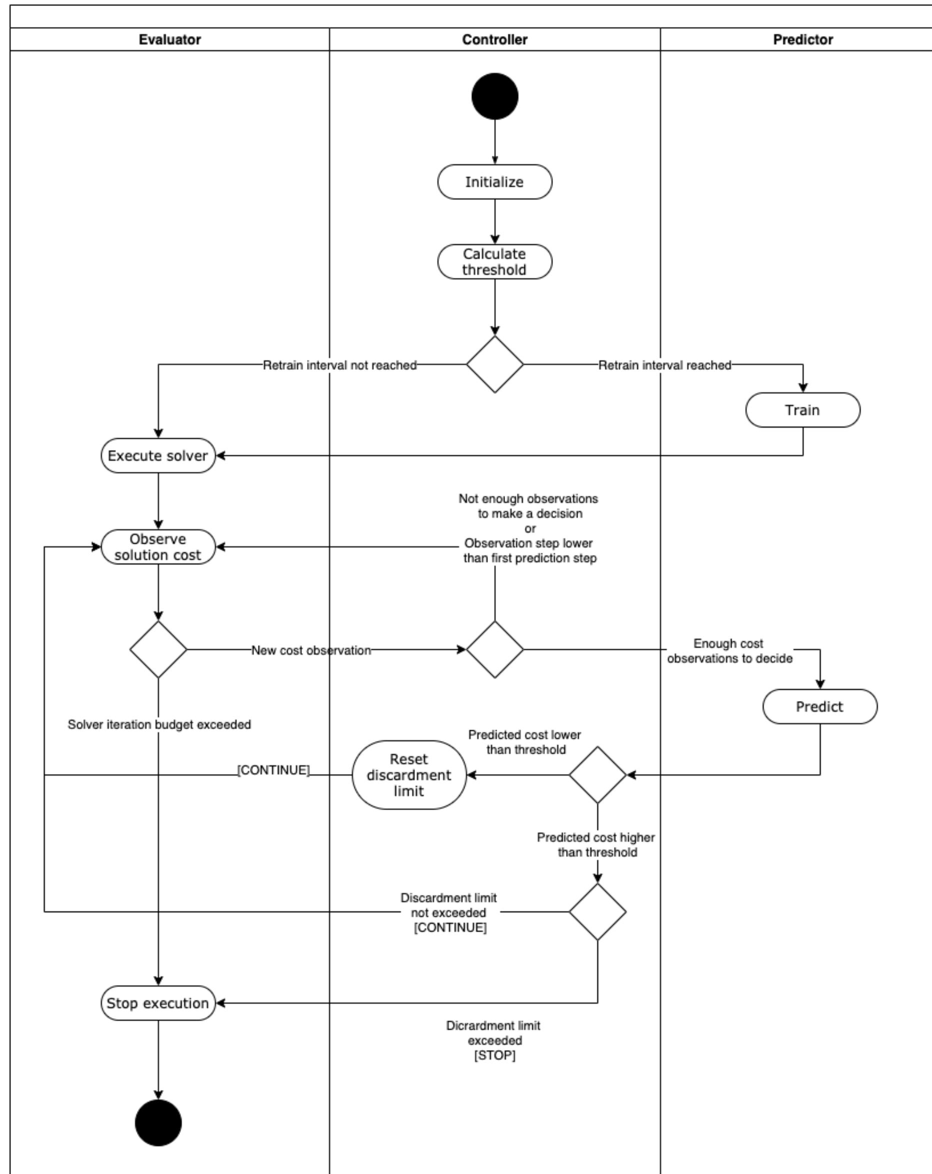
$$\sigma = \frac{1}{t} \sum_{i=1}^{t} c_i \tag{4.2}$$

where t is the threshold limit. Given the formal definition, threshold value is calculated as a mean of t minimal cost values observed in previous trials.

[5] Cost values at final step of the solver progress curve

Controller lifecycle and its communication with Evaluator and Predictor components is shown as activity diagram in Figure 4.3.

Figure 4.3: Controller lifecycle



Lifecycle starts in the beginning of the optimization trial. Controller is initialized, new threshold is calculated for the upcoming trial, and the underlying Predictor component is trained according to the retrain interval (Subsection 4.4). After the solver is executed by the Evaluator, new values of the solver progress curve are supplied for the Controller as the observations are made. Once Controller has enough data to make a prediction, it calls the Predictor component. After the final costs are predicted for each of the given problem instances, their mean value is being compared with the calculated threshold value and decision is made for the evaluation and sent to the Evaluator component.

Given predicted values $\boldsymbol{p} = \{p_1, p_2, ...p_k\}$ for k problem instances, optimization trials is considered unpromising for the condition $\overline{\boldsymbol{p}} > \sigma$.
Decision process is called repeatedly by the Evaluator component, according to the defined prediction limit variable. Configuration variables for the controller were designed in order to make its lifecycle flexible and easily adaptable to different cases.

4.5 Optimizer

Optimizer component, as the name suggests, has responsibility of running the hyperparameter optimization process for the solver (managed by Evaluator component). The Optimizer component runs the optimization process with the given optimization budget, for which it repeatedly executes optimization trials until the budget is exceeded. The optimization process consists of two main parts, described in the following subsections.

4.5.1 Initialization trials

To initialize the optimization process, a set of first m hyperparameter configurations[6] for the solver function is generated via a quasirandom Sobol sequence (Subsection 2.2.4). Quasirandom initial trials are used to obtain sufficient data about the solver hyperparameter space. Results from these trials are used in the next part of the optimization process to create the Surrogate model for the Bayesian optimization. Also, this initialization gives Evaluator component opportunity to collect data from the solver progress curve in order to initialize Early Stop policy.

4.5.2 Bayesian optimization

After the initial trials are finished, the Bayesian optimization model is initialized. The surrogate model is built upon the data obtained from the initial trials and new hyperparameter configuration is generated by acquisition function for the next trial. For the theory behind Bayesian optimization, refer to chapter 2.
Every time optimization trial is finished, the surrogate model is updated according to the cost obtained from the trial (returned by Evaluator) and a new configuration is generated. This process is called repeatedly until the given optimization budget is exceeded. Once the optimization process is finished, the configuration with the best

[6] Parameter m is configurable for the Optimizer component and should be chosen reasonably according to the dimension of the parameter configuration

result is derived from the Optimizer component and returned to the caller.

Chapter 5

Experiments

In this chapter, we evaluate the performance of our ESBO method through experiments. First, in section 5.1 we observe the outcome of combining our Early Stop policy with the Bayesian approach. Section 5.2 is dedicated to comparison of ESBO method with the classic Bayesian optimization. Finally, in section 5.3, method's ability to generalize for different problem instances is discussed and validated with experiments.

To unify the environment for the experiments, we fix the settings for the optimization methods. This includes defining the set of VRPTW problem instances, optimization budget for each method and the solver budget for each of the local search algorithms being experimented with. Fixing the iteration budgets for both optimizer and solver gives us opportunity to compare the methods in the matter of iterations being spared when the Early Stop policy decides to discard the optimization trial. Given this, we focus on how the increase in the amount of performed optimization trials affects the performance of the ESBO method. Fixed iteration budgets, problem instances, and other parameters for individual experiments are defined in the beginning of each section.

All experiments presented in this chapter were performed on Debian 6.3.0-18+deb9u1 machine with 16 cores / 32 threads, 256GB RAM, and 500GB SSD.

5.1 Early Stop policy analysis

We ran optimization process with multiple consecutive optimization trials controlled by our Early Stop policy in order to analyze how the prediction of the costs for optimization trials can affect the performance of underlying Bayesian optimization. In following subsections, we analyse the effect of Early Stop policy on the optimization process for each of the solvers.

For the experiments, we implemented simulated environment to track both the values predicted by the Early Stop policy, and the real values obtained from the solver execution for the full solver budget. Thus, during the experiments, if the optimization trial is chosen to be discarded, cost predicted at the point of discardment is stored and trial continues until it is finished. That way predicted cost can be compared with the cost of the solution the solver found at the final iteration point and the decision can be validated for the experiment. To simulate production scenario, actual values are only stored for the experiment purposes, whereas the predicted costs are the ones being used to update the surrogate function of BO model during the optimization process. In this section, we used fixed setting for each of the solvers prescribed in Table 5.1. Also, we fixed the configuration of the Controller component to the values prescribed in Table 5.2. Values for the Controller were chosen according to the empirical observations and logical judgments during the development of the ESBO method.

Solver	**Solver budget**	**Optimization trials**	**Problem instances**
TASP	10,000	50	C110_1, C110_2, C110_3
Jsprit	500	50	
OptaPlanner	100,000	40	

Table 5.1: Early Stop policy experiment settings

Variable	**Value**	
Discardment limit	1	
Retrain interval	5	
First prediction step	50%	70%
Prediction limit	5	
Threshold limit	5	

Table 5.2: Percentage value for the First prediction step variable describes the portion of steps in the solver progress curve, from which the controller is allowed to make predictions

5.1.1 TASP

In the Subfigure 5.1b, we see the Early Stop policy configured to start decision process at iteration 7,000 out of 10,000. This settings showed impressively good results with prediction values close to the real ones. Second Subfigure 5.1a describes the policy configured more aggresively, making decisions from iteration 5,000. We see how earlier decisions increased the error in predictions. However, predicted values still copy the

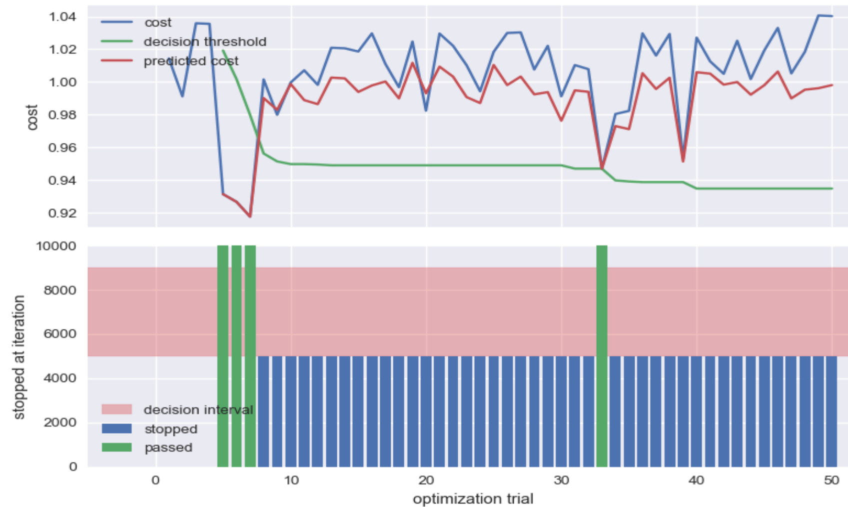
trend of the real values, identifying promising runs correctly.
Comparing these two Subfigures together, we can see that algorithm, given the decision threshold, let promising trials run until the final iteration and discarded the trials with unpromising cost predictions. Early stop policy seems to work in matter of deciding which runs are promising and which not for the TASP solver. Also, the fact that predictions underestimated the actual costs is beneficial because we rather want to let not promising runs finish than stop promising runs.

5.1.2 Jsprit

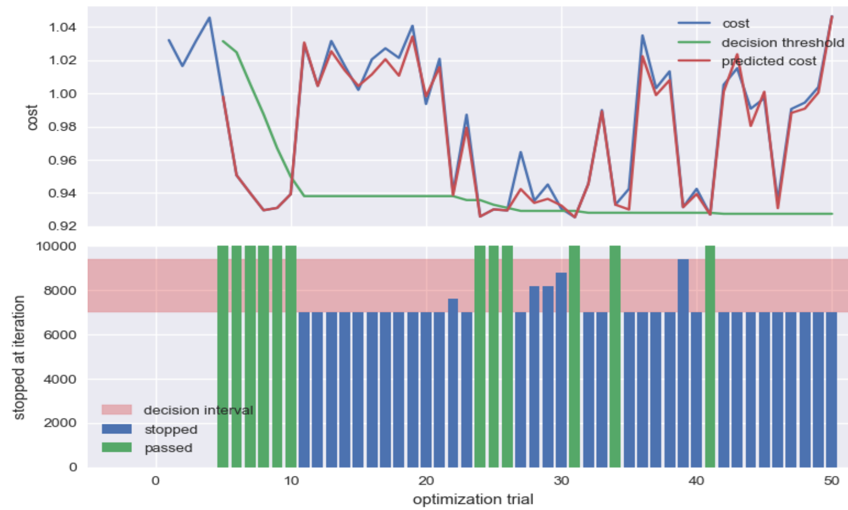
Early Stop policy did not perform as good for Jsprit solver as it did for TASP. As shown in Figure 5.2, values were predicted with significant error. From Subfigure 5.2a, it can be deduced, that with lower ”first prediction step” value (Subsection 4.4), policy has problem identifying the promising trials. Since trying to predict bigger time window can be impossible for the regression model, increasing the starting point of the prediction process is reasonable step to make the model more accurate. Increasing the first prediction step to iteration 7,000 resulted in lower error as shown in Subfigure 5.2b. However, according to the properties of the Jsprit solver (studied in Subection 3.4.1), there are obvious issues with integrating Early Stop policy into the solver evaluation, which should be studied further.

5.1.3 OptaPlanner

For OptaPlanner, we again observed significant prediction error for the setting of low ”first prediction step” value as shown in Subfigure 5.3a. In Subfigure 5.3b, with increased value of first prediction step, the increase in prediction accuracy is visible. However, policy still identifies only the subset of promising trials, which can lead to suppression of promising hyperparameter configurations for the optimization process.



(a)

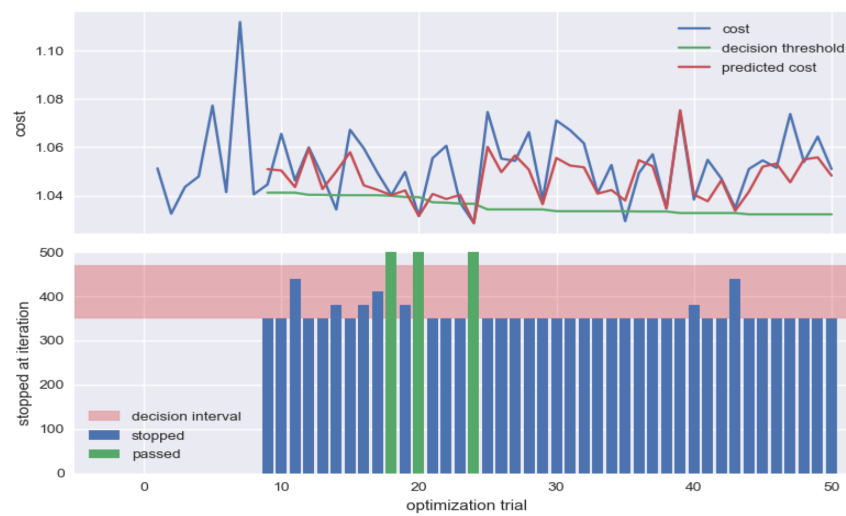


(b)

Figure 5.1: In the first graph (up) of each subfigure, the predicted and real cost is shown for each trial along with the threshold value, which was calculated at respective trial for the purpose of making a decision. Second graph (bottom) shows the decision process of the Early Stop policy, where the green bars prescribe the trials that were not stopped, and blue bars display discarded trials along with the iteration, at which the decision was made by the policy. Red area shows the interval, at which the policy was able to make decisions.

Figure 5.2: Early Stop policy analysis - Jsprit

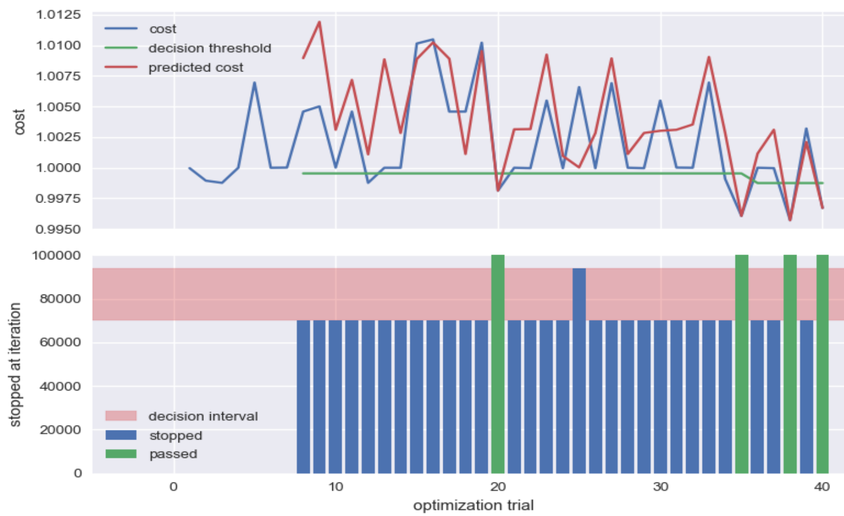
(a)



(b)

Figure 5.3: Early Stop policy analysis - OptaPlanner

(a)



(b)

5.1.4 Performance results

Due to the nature of the prediction task for Early Stop policy and its complexity, there is a clear dependence between performance of the policy and the time window[1] of the prediction. From the Figures 5.1, 5.2 and 5.3, we notice how the size of the time window can affect the prediction error for each solver. Therefore, setting the variable correctly according to the given solver can play crucial role in the performance and must be chosen reasonably to balance prediction error and amount of iterations being spared for the optimization process. Performance results of Early Stop policy

[1] Number of steps we want to predict

observed from our experiments are summarized in Table 5.3. For TASP solver, policy performed similarily good for both settings of "First prediction", identifying all of the promising trials. Jsprit and OptaPlanner showed to be more prone to the changes in prediction settings. Generally, for both solvers, increasing the "First prediction" value led to better performance with the loss in amount of iterations being spared for the optimizatin process.

Table 5.3: Early stop policy performance results

Experiment	**First prediction**	**Iterations spared**	**Promising trials detected**
TASP (b)	50 %	41.84 %	100 %
TASP (a)	70 %	18.96 %	100 %
Jsprit (a)	50 %	32.0 %	33.3 %
Jsprit (b)	70 %	21.9 %	60 %
OptaPlanner (a)	50 %	34.0	6.25%
OptaPlanner (b)	70 %	18.25 %	75 %

5.2 Comparison with Bayesian optimization

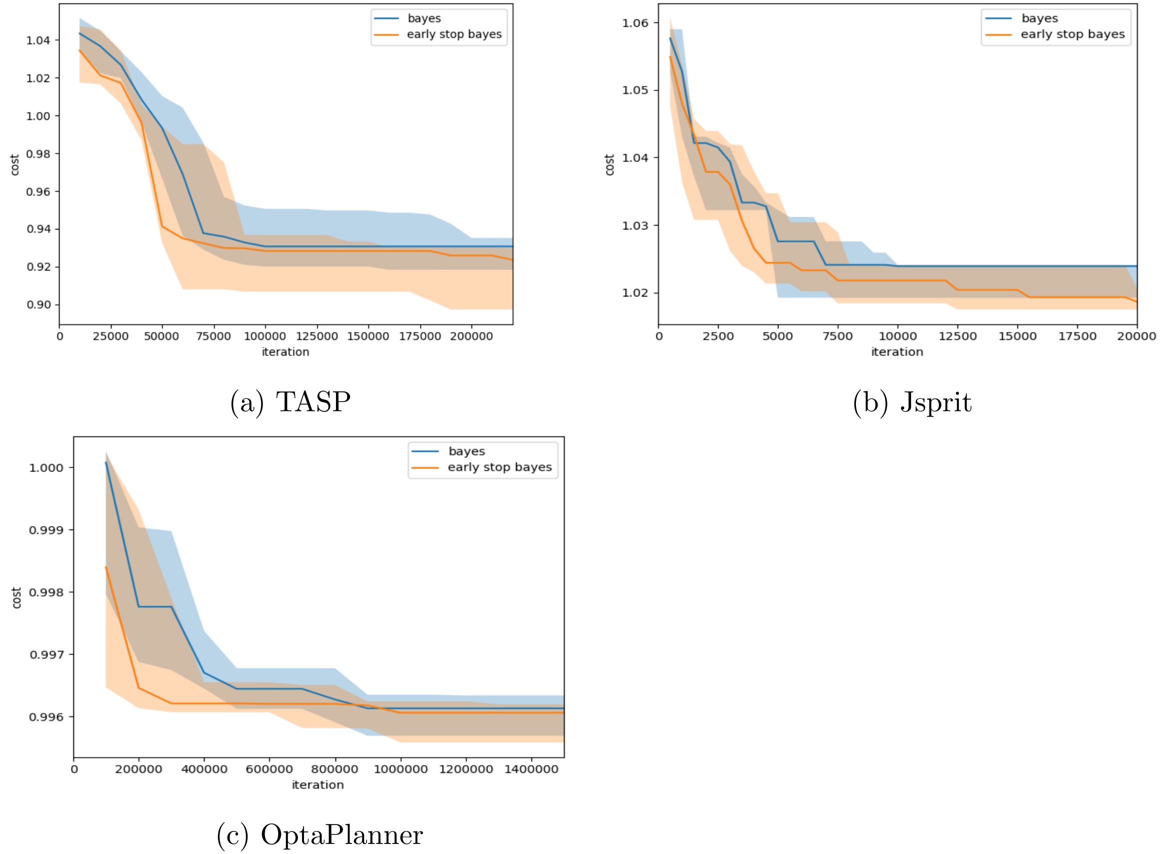
We performed comparison of our ESBO method with classic Bayesian optimization to analyze the improvement in the optimization process on selected solvers. Experiments were executed for all solvers with fixed setting prescribed in Table 5.4.

Table 5.4: Solver configurations used in experiments

Solver	**Solver budget**	**Optimizer budget**	**Problem instances**
TASP	10,000	220,000	C110_1, C110_2, C110_3
Jsprit	500	20,000	
OptaPlanner	100,000	1,500,000	

Twenty optimization processes were run for each pair of method and solver with the given setting. The comparison of the two methods for each solver is depicted in Figure 5.4. To describe the amount of improvement achieved by each optimization method, cost values were normalized by the same principle as defined in Equation 3.1 in Section 3.3.2.

For the TASP solver, ESBO method statistically outperformed classic Bayesian approach. As our method extends Bayesian method with Early Stop policy, it is visible how accelerating the optimization process led to faster convergence to the minimal value as shown in Subfigure 5.4a. This resulted in generally better hyperparameters found for the solver as can be derived from the area between 1st and 3rd quartile depicted in the Subfigure 5.4a.



(a) TASP

(b) Jsprit

(c) OptaPlanner

Figure 5.4: Blue and orange plots show optimization processes for each method. Every point on the plot represents median of cumulative minimum of each optimization process at the given trial. Filled areas display interval of 1st and 3rd quartile for each method.

When comparing methods for Jsprit solver, we can see notable improvement for the ESBO method, especially in the beginning of the optimization process, between 2,500 and 5,000 iterations. Nevertheless, the improvement is not of the same significance as for TASP solver. We assume the main cause is the bad predicability of the Jsprit solver execution, which can lead to wrong decisions done by Early Stop policy.
As can be denoted from the Subfigure 5.4b for Jsprit solver, neither of the methods was able to find better hyperparameter configuration for the given set of problem instances than the default one. Jsprit is, unlike TASP and OptaPlanner, not the general solver, but it is specifically designed to solve VRP problems. Thus, as we execute the solver on standard benchmark problem instances, we can expect that its parameters are already tunned for the problem domain. This fact makes Jsprit solver debatable as a subject of hyperparameter tunning. However, as we want to generalize the use of our optimization method, we ignore this observation, and focus on comparison of

inidividual optimization methods.

In results obtained for OptaPlanner, we see that the portion of improvement found by the optimization methods is smaller than for other solvers. This may be caused by the lower interconnection between the hyperparameters and the solver functionality (as studied in Chapter 3). Question arises, whether the bigger improvement is possible for the optimization in the given settings. However, with the given optimization budget, our method converged much faster (around iteration 200,000) than the Bayesian optimization (iteration 900,000), finding high quality configuration in $\frac{1}{5}$ of the budget needed by the Bayesian approach.

5.3 Generalization to unknown problem instances

To validate how the improvement obtained by our ESBO method is transferable onto different problem instances, we took the results obtained in comparison experiments (described in section 5.2) and ran the solvers with the winner configurations on the set of unknown problem instances. Fixed setting values can be seen in Table 5.5.

Table 5.5: Fixed settings for generalization experiments

Solver	**Solver budget**	**Problem instances**
TASP	10,000	C110_4, C110_5, C110_6
Jsprit	500	
OptaPlanner	100,000	

For each optimization method, set of winner parameter configurations is defined as a set of configurations that gave the best result in each of the optimization processes performed. Thus, for each pair of solver and optimization method, 20 winner configurations were picked for the experiment. Cost values obtained from the solver executions were normalized for each solver by dividing the original costs by the maximal cost obtained during the experiment for the given solver (identical normalization with maximal values was performed for experiments in Subsection 3.4.1).

As shown in Figure 5.5, we see that improvement of our methods against the classic Bayesian optimization was observed also for the data unknown to the optimization method. This confirms that the improvement obtained by our ESBO method persists when the resulting configurations are applied to the new, unseen problem instances. Specific results for each solver can be seen in Table 5.6.

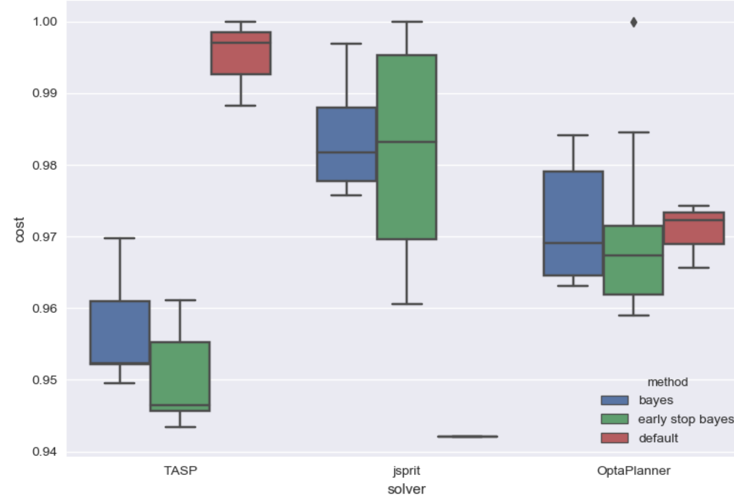


Figure 5.5: Results obtained by evaluating solvers with winner configurations. Boxplots show the distribution of the result costs for each method and default solver configuration (without optimization). Filled areas express the area between 1st and 3rd quartile. Horizontal lines for each boxplot show, from the bottom up, $1st\ quartile - 1.5 * median$, $median$, $3rd\ quartile + 1.5 * median$. Points display the outlier of the individual distributions

Solver	method	median	mean	min	max
TASP	ESBO	**0.946**	**0.949**	**0.943**	**0.961**
	Bayesian optimization	0.952	0.956	0.949	0.97
	Default	0.997	0.995	0.988	1.0
Jsprit	ESBO	0.983	0.981	0.960	1.0
	Bayesian optimization	0.982	0.984	0.976	0.997
	Default	**0.942**	**0.942**	**0.942**	**0.942**
OptaPlanner	ESBO	**0.967**	**0.970**	**0.959**	1.0
	Bayesian optimization	0.969	0.972	0.963	0.984
	Default	0.972	0.970	0.965	**0.974**

Table 5.6: In the Table, statistical values observed for each method and each solver are described. "Default" value in column "method" means that values were observed for default configuration of hyperparameters. Minimal values are highlighted per each solver

Generalization experiment showed, that ESBO method outperforms Bayesian optimization in most of the observed statistical metrics, when tested on unknown problem instances. Even though the difference is not significant, it is a convenient baseline for future experiments with the method. Also, despite the Jsprit solver (which was being discussed in Section 5.2), our method gave better results than default configurations for both TASP and OptaPlanner.

Chapter 6

Conclusion

Local search algorithms are typically configurable by the number of hyperparameters that impact their behavior and performance. However, setting correct parameter values for the algorithm is often complicated by the lack of documentation on their meaning. Also, the expensiveness of the algorithm's evaluation prevents the use of standard hyperparameter tunning methods like grid search or random search when the time budget is limited, especially for the hyperparameters of higher dimensionality.

We studied the Bayesian optimization method as a promising tool for hyperparameter optimization of local search algorithms. By comparison with the random-search method, Bayesian approach proved to be more efficient and suitable for the optimization task.

In default setting, Bayesian optimization treats the objective function as a black-box. This property restricts it from using the knowledge about the domain, if available. Therefore, we studied the possible adaptation of the Bayesian method to our domain, and designed the modified optimization framework, which we named Early Stop Bayesian optimization. Our modification uses Random Forest regression to extrapolate the curve of the local search algorithm's internal progress. The selective policy then uses the extrapolation results to stop unpromising evaluations of the algorithm. The integration of such policy leads the search of the optimization method towards the promising hyperparameter configurations, not wasting resources on unpromising ones.

We tested our method on the set of three local search algorithm implementations. In performed experiments, our modification showed generally better results and faster convergence to the high-quality configuration than the classic Bayesian method.

6.1 Future work

Since our method directly relies on Random Forest regression's performance, we see the opportunity in broader research of the time-series forecasting techniques. That includes the study of new, more descriptive feature variables for the time-series, and also research of other machine learning algorithms that have potential in the time-series prediction, like neural networks. If the accuracy of predictions would increase by incorporating new feature variables and techniques, the ESBO method would possibly perform better and would be able to converge even faster to the high-quality hyperparameter configuration without providing noisy objective values for the Bayesian optimization model.
Another improvement could be made by setting a different configuration for Bayesian optimization. In our work, we used the Gaussian process as a surrogate model and Noisy Expected improvement as an acquisition function for the Bayesian technique. However, many other approaches exist, that could be more suitable for our case and has not been tested for the domain of hyperparameter tunning of the local search algorithm.

Bibliography

[1] J. Mockus, V. Tiesis, and A. Zilinskas, *The application of Bayesian methods for seeking the extremum*, 09 2014, vol. 2, pp. 117–129.

[2] J. Snoek, H. Larochelle, and R. P. Adams, "Practical Bayesian Optimization of Machine Learning Algorithms," *arXiv e-prints*, p. arXiv:1206.2944, Jun. 2012.

[3] J. Snoek, O. Rippel, K. Swersky, R. Kiros, N. Satish, N. Sundaram, M. M. A. Patwary, Prabhat, and R. P. Adams, "Scalable Bayesian Optimization Using Deep Neural Networks," *arXiv e-prints*, p. arXiv:1502.05700, Feb. 2015.

[4] C. E. Rasmussen and C. K. I. Williams, *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2005.

[5] B. Letham, B. Karrer, G. Ottoni, and E. Bakshy, "Constrained Bayesian Optimization with Noisy Experiments," *arXiv e-prints*, p. arXiv:1706.07094, Jun. 2017.

[6] V. Picheny, D. Ginsbourger, Y. Richet, and G. Caplin, "Quantile-based optimization of noisy computer experiments with tunable precision," *Technometrics*, vol. 55, no. 1, pp. 2–13, 2013. [Online]. Available: https://doi.org/10.1080/00401706.2012.707580

[7] R. Harman and V. Lacko, "On decompositional algorithms for uniform sampling from n-spheres and n-balls," *Journal of Multivariate Analysis*, vol. 101, no. 10, pp. 2297 – 2304, 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0047259X10001211

[8] P. Bratley and B. L. Fox, "Algorithm 659: Implementing sobol's quasirandom sequence generator," *ACM Trans. Math. Softw.*, vol. 14, no. 1, p. 88–100, Mar. 1988. [Online]. Available: https://doi.org/10.1145/42288.214372

[9] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *J. Mach. Learn. Res.*, vol. 13, no. null, p. 281–305, Feb. 2012.

[10] B. Solutions. Tasp. https://bitbucket.org/blindspotsolutions/tasp-example/src/master/.

[11] Blindspot solutions. http://blindspot.ai/.

[12] GraphHopper. graphhopper/jsprit. https://github.com/graphhopper/jsprit.

[13] KIE. kiegroup/optaplanner. https://github.com/kiegroup/optaplanner.

[14] (2008) VRPTW benchmark - 1000 customers. https://www.sintef.no/projectweb/top/vrptw/homberger-benchmark/1000-customers/.

[15] Facebook. facebook/ax. https://github.com/facebook/Ax.

[16] M. Balandat, B. Karrer, D. R. Jiang, S. Daulton, B. Letham, A. G. Wilson, and E. Bakshy, "Botorch: Programmable bayesian optimization in pytorch," 2019.

[17] Facebook. facebook/botorch. https://github.com/pytorch/botorch.

[18] A. Meyler, G. Kenny, and T. Quinn, "Forecasting irish inflation using arima models," 1998.

[19] V. Ediger and S. Akar, "Arima forecasting of primary energy demand by fuel in turkey," *Energy Policy*, vol. 35, no. 3, pp. 1701 – 1708, 2007. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0301421506002291

[20] L. Breiman, "Machine learning, volume 45, number 1 - springerlink," *Machine Learning*, vol. 45, pp. 5–32, 10 2001.

[21] H. Tyralis and G. Papacharalampous, "Variable selection in time series forecasting using random forests," *Algorithms*, vol. 10, no. 4, p. 114, Oct 2017. [Online]. Available: http://dx.doi.org/10.3390/a10040114

[22] M. Kumar and T. M., "Forecasting stock index movement: A comparison of support vector machines and random forest," *SSRN Electronic Journal*, 01 2006.

[23] K.-j. Kim and I. Han, "Genetic algorithms approach to feature discretization in artificial neural networks for the prediction of stock price index," *Expert Systems with Applications*, vol. 19, pp. 125–132, 08 2000.

[24] 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*, vol. 12, pp. 2825–2830, 2011.

Appendix A

Attached files

The attached files contain the electronic version of this thesis, source files for this thesis, source code of our optimization framework and Gehring Homberger benchmark instance files that were used in our experiments.

- `thesis.pdf` ...... Electronic version of the thesis
- `src` ...... Folder with source files
  - `esbo` ...... Folder with source codes for the optimization framework
  - `thesis` ...... Folder with thesis source files
- `benchmarks` ...... Folder with benchmark problem instances