Automated machine learning with gradient boosting and meta-learning

van Hoof, J.M.A.P.

Award date:
2019

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Department of Mathematics and Computer Science
Architecture of Information Systems Research Group

Automated machine learning with gradient boosting and meta-learning

Master Thesis

J.M.A.P. van Hoof

Supervisors:
Dr. ir. J. Vanschoren

Revised version

Eindhoven, December 2018

Abstract

The success of machine learning in a broad range of applications has led to an ever-growing demand for machine learning systems that can be used off the shelf by non-experts. To be effective in practice, such systems need to automatically choose a good algorithm and feature preprocessing steps for a new dataset at hand, and also set their respective hyperparameters. Recent work has started to tackle this automated machine learning (AutoML) problem with the help of efficient Bayesian optimization methods.

In this work we introduce several improvements over the existing automated machine learning methods. We propose to improve Bayesian Optimization with gradient boosted decision trees using quantile regression. We furthermore discuss a running time estimation component that estimates running time better than existing techniques, which can help speed up the Bayesian Optimization process in some cases.

We furthermore propose an efficient meta-learner, again based on gradient boosted decision trees, that is able to effectively predict the ranking order of the performance of each algorithm configuration on an unseen task. We then use this meta-learner together with a surrogate model to improve the Bayesian Optimization process even further.

Finally, we explore algorithm selection as a multi-armed bandit problem using a range of techniques from the multi-armed bandit problem space.

Preface

This thesis is the result of the graduation project of the Computer Science and Engineering program at the Eindhoven University of Technology (TU/e). The research of this project is performed within the Data Mining Group of the TU/e in collaboration with OpenML.

I would like to thank my supervisor Joaquin Vanschoren. He provided useful directions and feedback to my project. I would also like to thank Matthias Fuehrer, the creator of AutoSklearn, for his help and support on AutoSklearn and the OpenML API, and Randal S. Olson, the creator of TPOT, for his help and support with TPOT. Furthermore, I am grateful for the opportunity to make use of the compute servers from the TU/e and SurfSara.

I would like to thank my colleagues, friends and family for their support during my studies.

J.M.A.P. (Jeroen) van Hoof
Eindhoven, the Netherlands
September 2018

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

We will first give a motivation for automated machine learning in section 1.1. Then we set our goals and research scope in section 1.2. After that, we state our research problem in section 1.3. And finally, we give an overview of the chapters in section 1.4.

1.1 Motivation

Machine learning is commonly described as a field of study that gives computers the ability to learn without being explicitly programmed. Despite this common claim, machine learning practitioners know that designing effective machine learning pipelines is often a tedious endeavor, and requires considerable experience with machine learning algorithms.

Machine learning relies heavily on choosing the right algorithm and good parameters for that algorithm. Even experienced machine learning practitioners do not know exactly which algorithms and parameters will perform the best for a particular dataset, and often end up in a local optimum. Furthermore, it requires a lot of manual programming and time to try out configurations.

Automated machine learning tries to solve this problem. It was proposed as a solution to the ever-growing challenge of applying machine learning. Automating the end-to-end process of applying machine learning offers the advantages of faster creation of those solutions and models that often outperform models that were optimized by hand.

1.2 Goals

The goal of this work is to improve upon the current state-of-the-art automated machine learning tools. We will use AutoSklearn [9] as our baseline state-of-the-art AutoML tool to compare against. AutoSklearn improved on existing AutoML methods by automatically taking into account past performance on similar datasets (meta-learning), and by constructing ensembles from the models evaluated during the optimization (ensemble construction).

The notion of "existing AutoML methods" here, refers to SMAC [13], the underlying tool that AutoSklearn is based on. SMAC is a tool for optimizing the hyperparameters of machine-learning algorithms (hyperparameter optimization). To automate both algorithm selection and hyperparameter optimization, AutoSklearn reduces this combined problem to a hyperparameter optimization problem.

In this project, we will limit our research to algorithm selection, hyperparameter optimization and meta-learning.

1.3 Problem statement

In subsection 1.3.1 we give a formalization of algorithm selection and hyperparameter optimization combined as one problem, and in subsection 1.3.2 we state our research problem.

1.3.1 CASH

Automating the process of selecting an algorithm and optimizing its hyperparameters can be formalized as the Combined Algorithm Selection and Hyperparameter optimization (CASH) problem, as described by Feurer et al. [9]:

Definition 1: *Let $\mathcal{A} = \{A^{(1)}, ..., A^{(R)}\}$ be a set of algorithms, and let the parameters of each algorithm $A^{(j)}$ have domain $\Lambda^{(j)}$. Further, let $D_{train} = \{(x_1, y_1), ..., (x_n, y_n)\}$ be a training set which is split into K cross-validation folds $\{D^{(1)}_{valid}, ..., D^{(K)}_{valid}\}$ and $\{D^{(1)}_{train}, ..., D^{(K)}_{train}\}$ such that $D^{(i)}_{train} = D_{train} \setminus D^{(i)}_{valid}$ for $i = 1, ..., K$. Finally, let $\mathcal{L}(A^{(i)}_{\boldsymbol{\lambda}}, D^{(i)}_{train}, D^{(i)}_{valid})$ denote the loss that algorithm $A^{(j)}$ achieves on $D^{(i)}_{valid}$ when trained on $D^{(i)}_{train}$ with hyperparameters $\boldsymbol{\lambda}$. Then, the Combined Algorithm Selection and Hyperparameter optimization (CASH) problem is to find the joint algorithm and hyperparameter setting that minimizes this loss:*

$$A^*, \boldsymbol{\lambda}_* \in \underset{A^j \in \mathcal{A}, \lambda \in \Lambda^j}{\arg\min} \frac{1}{K} \sum_{i=i}^{K} \mathcal{L}(A^{(i)}_{\boldsymbol{\lambda}}, D^{(i)}_{train}, D^{(i)}_{valid})$$

1.3.2 Research problem

Our research question will read as follows: **"How can we further improve the performance of current Bayesian Optimization approaches for automated machine learning?"**.

- **Hyperparameter optimization:** We want to find a method that scales well with the number of iterations, and finds high-performing settings that are preferably also quick to execute. Furthermore, it is also desirable that our method works well with a small number of iterations as Bayesian Optimization gathers more knowledge over time and bases its knowledge on previously gathered knowledge.

- **Meta-learning:** Using OpenML's database of experiments, we strive to create a meta-model that predicts the performance of an algorithms' setting on a task with certain properties (meta-features). More specifically, we want a meta-model that is able to distinguish between high-performing settings and low-performing settings. We then make use of this meta-model to warm-start Bayesian Optimization.

  We differ here from AutoSklearn's method, which only saves the best instantiation (an algorithm with a specific configuration) per task, and then uses these instantiations to warm-start Bayesian Optimization.

- **Algorithm selection:** Normally, Bayesian Optimization (BO) optimizes a single machine learning algorithm. To extend this optimization process from single-model optimization to multi-model optimization, we will create several competing BO-processes, where each process is tied to a different machine learning algorithm. We then run a competition process with multiple rounds, where we decide per round whose turn it is, i.e. which BO-process can execute one iteration. We want to allocate these rounds strategically, such that we allocate more rounds to BO-processes that are likely to eventually produce the highest performing machine learning instantiation. We will explore this problem of allocating rounds as a multi-armed bandit problem.

  We differ here from AutoSklearn, which simply reduces the CASH-problem to a hyperparameter optimization problem.

1.4 Thesis outline

The preliminaries, chapter 2, give an introduction into the topics of hyperparameter optimization, meta-learning, the multi-armed bandit problem and current automated machine learning tools. Additionally, we show various gradient boosting techniques and implementations such as XGBoost, LightGBM and Catboost. And we also explain some important concepts of OpenML.

Next, in chapter 3 we discuss two problems that need to be overcome before we can continue with Bayesian Optimization methods and meta-learning. In section 3.1 we discuss the required preprocessing steps to split and transform hyperparameter values of hyperparameters that accept a variety of different value types. Furthermore, in section 3.2 we explain two approaches to modify modify Random Forests such that they can express uncertainty about their estimates.

We bundled and compressed our early experiments related to Bayesian Optimization into chapter 4, where we describe a small part of these experiments. This chapter mainly explains the differences in performance and running time of Gaussian Processes, Randomized Search and Tree-based models.

In chapter 5, we research and optimize more efficient methods for estimating running time and performance for different algorithm configurations. Later, we combine these methods and finally test our methods on a larger set of datasets using a different target model. In this chapter, we propose two new improvements upon current Bayesian Optimization techniques.

Then, chapter 6 describes how we can use the performance of algorithm runs on other datasets to create a predictive model that is able to predict the performance of a configuration on a new dataset. We then describe how we can use such a meta-learner to speed up Bayesian Optimization.

In chapter 7, we test different algorithms from the multi-armed bandit problem space, to explore whether these algorithms are suitable for allocating resources towards optimizing competing algorithms and how well these algorithms perform in this task.

Finally, in chapter 8 we draw the main take-away conclusions from our research and give ideas for future work.

Chapter 2

Preliminaries

In this chapter we will describe the building blocks on which the thesis builds. In section 2.1 we will describe the OpenML platform and the concepts that are important to understand. In section 2.2 and section 2.3 we describe parameter optimization and meta-learning, and in section 2.4 we describe the multi-armed bandit problem and its relation the algorithm-selection part of the CASH-problem. These components together form a complete approach to the CASH-problem. Finally, in section 2.5 we describe two popular state-of-the-art approaches towards the CASH-problem.

In our experiments, we mainly use machine learning algorithms from the Scikit-learn framework. We extend our set of machine learning algorithms with popular gradient boosting algorithms, which we will discuss in section 2.6.

2.1 OpenML

OpenML is an open platform for sharing machine learning knowledge and research. It allows everyone to share interesting datasets, analyse data, and work on solutions together. This makes one's work more visible, reusable, and easily citable.

The OpenML integrations make sure that all uploaded results are linked to the exact (versions) of datasets, workflows, software, and the people involved. The predictions are generated locally using exact procedures, and these are evaluated server-side so that results are directly comparable and reusable in further work. Wherever possible, OpenML extracts clear descriptions of machine learning workflows and models.

From an automated machine learning point of view, OpenML is very interesting because of the following reasons:

- There are plenty of datasets that are all in the same format, and these can be downloaded automatically via the OpenML API. OpenML provides groups of datasets (studies) that are considered to be good for experiments (not too hard and not too easy).
- Millions of runs have already been executed and uploaded to OpenML. And because all runs are reproducible, we can inspect these and learn about the algorithm's performance in combination with its hyperparameters and the task performed.
- We can easily compare our results against existing results on OpenML, which can serve as a baseline. We can experiment with different (automated) machine learning algorithms and compare their performance.

2.1.1 Concepts

OpenML operates on a number of core concepts which are important to understand:

- Datasets: Datasets are pretty straight-forward. They simply consist of a number of rows, also called instances, usually in tabular form.
- Tasks: A task consists of a dataset, together with a machine learning task to perform, such as classification or clustering and an evaluation method. For supervised tasks, this also specifies the target column in the data.
- Flows: A flow identifies a particular machine learning algorithm or pipeline from a particular library or framework such as Weka, MLR or Scikit-learn.
- Runs: A run is a particular flow, that is algorithm, with a particular parameter setting, applied to a particular task.

2.2 Hyperparameter optimization

Hyperparameter optimization is the problem of choosing a set of optimal hyperparameters for a learning algorithm. The same kind of machine learning model can require different constraints, weights or learning rates to generalize different data patterns. These measures are called hyperparameters, and have to be tuned so that the model can optimally solve the machine learning problem. Hyperparameter optimization finds a tuple of hyperparameters that yields an optimal model which minimizes a predefined loss function on given independent data. Cross-validation is often used to estimate this generalization performance.

2.2.1 Grid Search

The simplest approach to determine the best parameters for an algorithm is via a Grid Search. Grid Search needs a set of pre-specified parameters and will try out every possible combination of these parameters. This is a brute-force approach and because we go over each combination one-by-one, thus moving linearly through regions of the parameter space, it is not possible to make a trade-off between computation time and performance.

2.2.2 Randomized Search

Randomized Search draws random samples from a specified configuration space. This allows the user to specify continuous spaces to draw samples from, rather than to choose discrete values. An advantage of Randomized Search is that it can easily be parallized: we can start the same algorithm on two compute nodes and both instances will start drawing random samples and execute them. in the literature it is conventional to use Randomized Search as a baseline. Frequently there is one normal run of Randomized Search, and one “Randomized Search 2x” run included, which is executed in parallel on two nodes.

2.2.3 Bayesian Optimization

Bayesian optimization fits a probabilistic model to capture the relationship between hyperparameter settings and their measured performance. The goal of Bayesian Optimization is to find an approximate minimum to some function that is expensive to evaluate. In this case the objective function is the algorithm that we are trying to optimize, and the function’s parameters are the hyperparameters of the algorithm.

The Bayesian optimization technique creates a prior over the objective function and combines it with evidence to get the posterior. This allows for a utility-based selection for the next observation to make on the objective function, which must take into account both exploration (sampling from areas of high uncertainty) and exploitation (sampling areas likely to offer improvement over the current best observation).

The model used for approximating the objective function is called a surrogate model. While Bayesian optimization with Gaussian process models as the surrogate model perform the best in

low-dimensional problems with numerical hyperparameters, probabilistic tree-based models have been shown to be more successful in high dimensional structured and partly discrete problems [7].

Typically a Gaussian process is used to build the probabilistic model of the objective function. And to determine the next most useful observation to make on the objective function, an acquisition function measures the expected utility of performing an evaluation of the objective at a new point.

2.2.4 Expected Improvement

The Expected Improvement (EI) is a popular acquisition function. Although it is commonly known to be too greedy, it is widely used due to its simplicity and ability to handle uncertainty and noise. Figure 2.1 demonstrates how the EI acquisition function chooses a next sample to evaluate.

The expected improvement approach states how much improvement I of the maximum function value of the surrogate model, f^*, is going to be expected according to the probability distribution of the function values. This improvement is defined as follows:

$$I(x) = \max(Y - f^*, 0) \tag{2.1}$$

where Y is the random variable $\sim \mathcal{N}(\mu, \sigma^2)$ that corresponds to the function value at x. And because I is then also a random variable, we can consider the average (expected) improvement to assess x:

$$EI(x) = E_{Y \sim \mathcal{N}(\mu, \sigma^2)}[I(x)] \tag{2.2}$$

With the reparameterization trick, $Y = \mu + \sigma\varepsilon$ where $\varepsilon \sim \mathcal{N}(0, 1)$, we have:

$$EI(x) = E_{\varepsilon \sim \mathcal{N}(0,1)}[I(x)] \tag{2.3}$$

This function can then be rewritten (from linearity of integral, and the definition of $\frac{d}{d\varepsilon} e^{-\varepsilon^2/2}$ derivative) to its closed form:

$$EI(x) = \int_{-\infty}^{\infty} I(x)\phi(\varepsilon)d\varepsilon \tag{2.4}$$

$$EI(x) = \int_{-\infty}^{(\mu - f^*)/\sigma} (\mu + \sigma\varepsilon - f^*)\phi(\varepsilon)d\varepsilon \tag{2.5}$$

$$EI(x) = (\mu - f^*)\Phi(\frac{\mu - f^*}{\sigma}) - \sigma \int_{-\infty}^{(\mu - f^*)/\sigma} \varepsilon\phi(\varepsilon)d\varepsilon \tag{2.6}$$

$$EI(x) = (\mu - f^*)\Phi(\frac{\mu - f^*}{\sigma}) + \frac{\sigma}{\sqrt{2\pi}} \int_{-\infty}^{(\mu - f^*)/\sigma} (-\varepsilon)e^{-\varepsilon^2/2}d\varepsilon \tag{2.7}$$

$$EI(x) = (\mu - f^*)\Phi(\frac{\mu - f^*}{\sigma}) + \frac{\sigma}{\sqrt{2\pi}} e^{-\varepsilon^2/2}|_{-\infty}^{(\mu - f^*)/\sigma} \tag{2.8}$$

$$EI(x) = (\mu - f^*)\Phi(\frac{\mu - f^*}{\sigma}) + \sigma\big(\phi(\frac{\mu - f^*}{\sigma}) - 0\big) \tag{2.9}$$

$$EI(x) = (\mu - f^*)\Phi(\frac{\mu - f^*}{\sigma}) + \sigma\phi(\frac{\mu - f^*}{\sigma}) \tag{2.10}$$

Here, ϕ and Φ are the PDF and CDF of the standard normal distribution, respectively.

2.2.5 Expected improvement per second

Different regions of the parameter space may result in vastly different execution times, due to varying regularization, learning rates, etc. To improve our performance in terms of wallclock

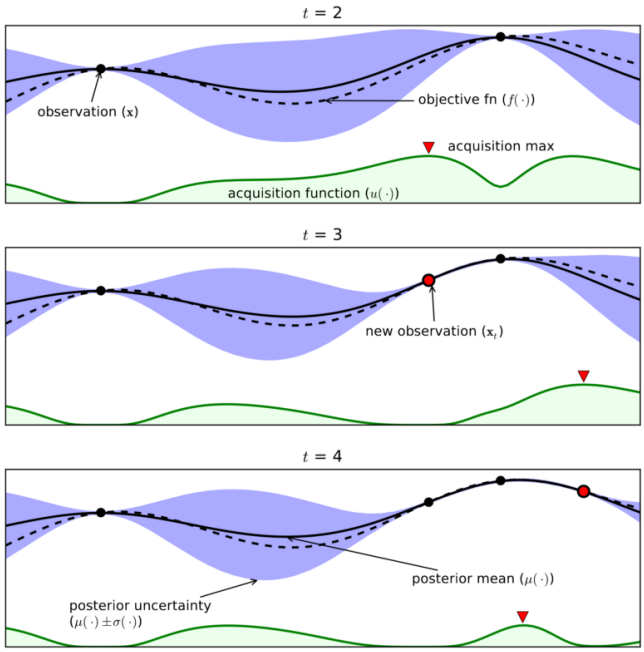


Figure 2.1: An example with three iterations of Bayesian optimization [4].

time, Snoek et al. [22] proposes an extension to EI: *expected improvement per second* (EI/s), which prefers to acquire points that are not only likely to be good, but that are also likely to be evaluated quickly.

The name is slightly misleading, as we are not calculating the expected improvement *per second*, but rather per *logarithmic transformed second*. More formally, given a configuration $\theta \in \Theta$, we define our duration function $c(\theta) : \Theta \to \mathbb{R}^+$, and we want to estimate $\ln c(\theta)$.
We show an example of using Bayesian optimization in Figure 2.1. The figure visualizes three iterations (denoted by t) of Bayesian Optimization. The independent variable (the x-axis) is the value for the hyper-parameter (e.g. learning rate) we are optimizing. For each new value we try out, we observe its result (the y-axis) and our surrogate model interpolates between these points. The acquisition function is then inferred from the interpolated mean and the posterior uncertainty, and we select or next value at the acquisition maximum.

If an observation shows worse results than another observation, the surrounding area of the worst observation is less interesting. Furthermore, if we have not made any observations in a certain area, this area will be more interesting, because the posterior uncertainty is high.

2.2.6 Gaussian Processes

The goal of regression is to find an underlying function $f(x)$ of some data. If we expect this function to be linear, we want to fit a line $f(x) = \theta_0 + \theta_1 x$ where θ_0 and θ_1 are parameters we want to optimize, with for example least-squares (linear regression). Or, if we suspect $f(x)$ to be quadratic ($f(x) = \theta_0 + \theta_1 x + \theta_2 x^2$), cubic or even non-polynomial, we can add more parameters.

However, say we do not want to specify upfront how many parameters are involved. We would like like to consider every possible function that matches our data, with however many parameters are involved. That is, we want a non-parametric solution.

Gaussian Processes (GP) defines a distribution on a set of functions. GP takes the prior distribution of possible base functions and updates this as data points are observed, producing the posterior distribution over functions, see Figure 2.2 [1]. In other words, we observe some points and

[1] https://commons.wikimedia.org/wiki/File:Gaussian_Process_Regression.png

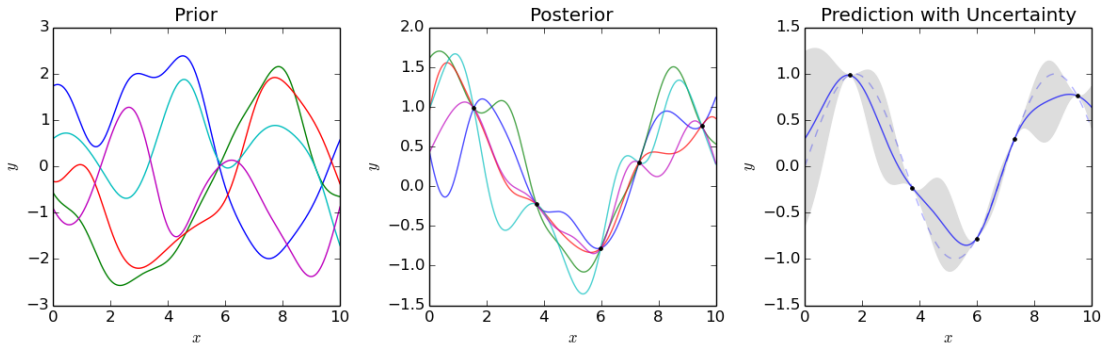


Figure 2.2: Gaussian Process Regression (prediction) with a squared exponential kernel. Left plot are draws from the prior function distribution. Middle are draws from the posterior. Right is mean prediction with one standard deviation shaded.

want to assign probabilities to each function that could be drawn through those points. The hope is that the functions with the highest probability are very similar to the underlying true function.

Then the question is how to assign probability to each function. GPs parameterize the probability in terms of a $n \times n$ co-variance matrix: a distance function is calculated for every pair of the n total observed points. All this matrix does is control the GP's preference for smoothness: it ensures that values that are close together in input space will produce output values that are close together. Figure 2.3 shows some common covariance functions.

The prior's co-variance is specified by a passing a kernel. Figure 2.4 shows the effect of choosing different kernels. The hyperparameters of the kernel are optimized during training, using an optimizer, by maximizing the log-marginal-likelihood (LML). By default, the L-BFGS-B algorithm [27] is used as optimizer. As the LML may have multiple local optima, the optimizer can be started repeatedly to avoid ending up in an unfavorable local optimum.

One important characteristic of a kernel is the length-scale. The length-scale of the process can be intuitively understand as "how close" two points x and x' have to be to influence each other significantly. Figure 2.5 shows the effect of choosing different values for the length scale.

Figure 2.6 visualizes how the GP learns/updates the probabilities of possible base functions based on new data. When using GP for Bayesian Optimization, every observed data point can be viewed as an iteration. The red line can be seen as the posterior mean shown in figure Figure 2.1, and when the spread of function lines is wider, we have more uncertainty about this region, which is similar to the posterior uncertainty.

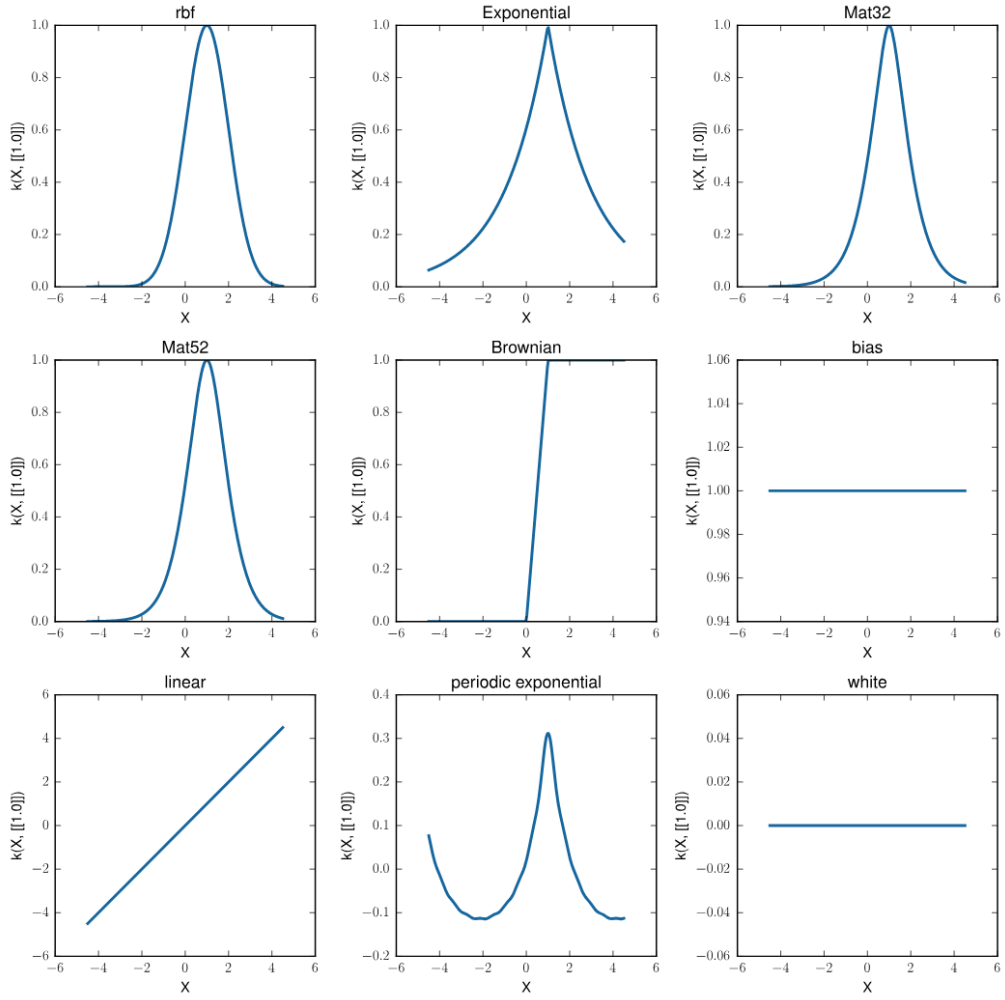


Figure 2.3: Common covariance functions

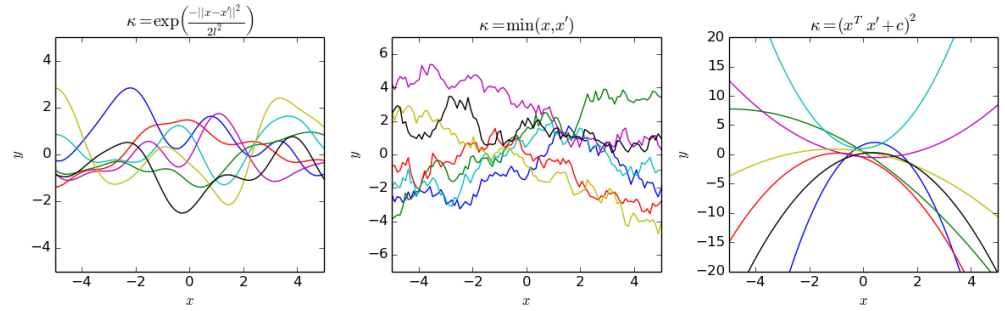


Figure 2.4: The effect of choosing different kernels on the prior function distribution of the Gaussian process. Left is a squared exponential kernel. Middle is Brownian. Right is quadratic.

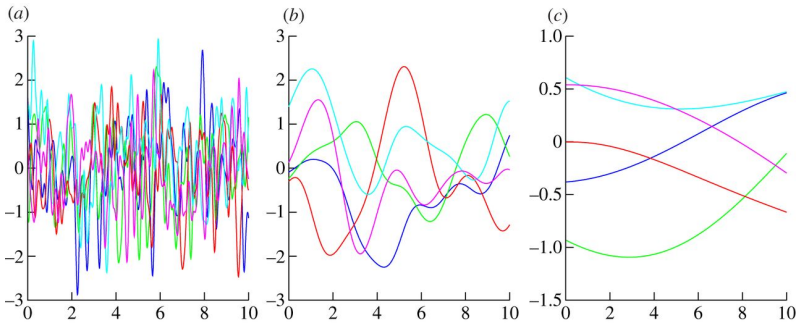


Figure 2.5: Functions drawn from a GP with a squared exponential covariance function with output scale $h = 1$ and length scales $\lambda = 0.1$ (a), 1 (b), 10 (c).

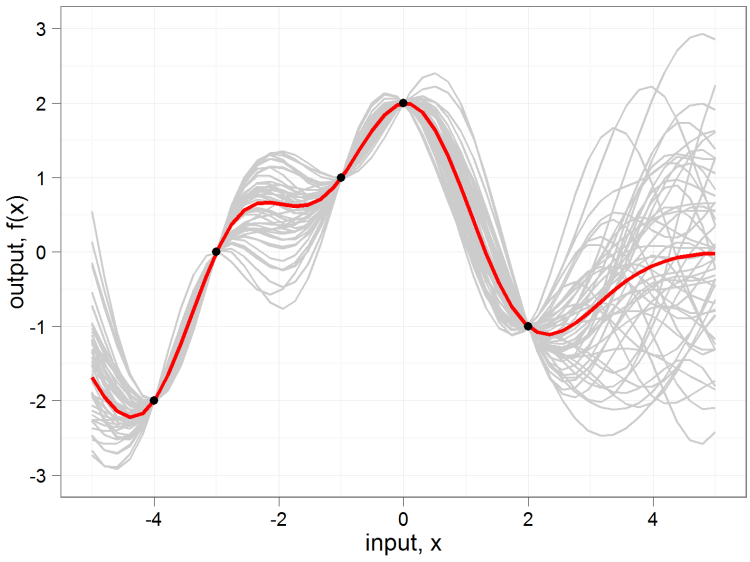


Figure 2.6: A visualization of Gaussian Processes.

2.3 Meta-learning

One aspect of Meta-learning is to learn about how machine learning algorithms perform across a range of tasks. It aims to learn which algorithm with which hyperparameter setting will work well for a dataset with certain characteristics.

If a meta-model can predict what data characteristics make a certain machine learning algorithm work well, it can be used to prune the search space. It can immediately suggest potential good configurations or discard bad ones. In subsection 2.3.2 we elaborate on that topic. Meta-learning is applied to perform algorithm selection, as well as hyperparameter optimization.

In order to train a meta-model, a meta-dataset from which to learn is required. A meta-dataset contains information about machine learning experiments. For instance, it captures which machine learning algorithm is used, with which hyperparameter configurations, and how well the resulting model performed. Additionally, for each such experiment it also describes the dataset on which the experiment was performed. The description of the dataset is done by meta-features that capture information about the data, such as the number of attributes, percentage of numerical features or the accuracy of simple classifiers.

2.3.1 Meta features

To create an effective meta-model, the meta-features used have to describe the datasets well. The features should be good predictors of the relative performance of algorithms. Several different types of meta-features have been developed, ranging from simple features such as the number of samples in a dataset, to more complex ones. Most meta-features fall into one of the following groups [3, 19, 25]:

Simple, Statistical and Information-theoretic

These features sometimes appear separately, as they have mixed origins. Simple features are quick to extract, such as the number of samples in the dataset, or the proportion of samples in the majority class. Statistical features include information about the numeric features, such as the mean skewness of all features. Finally, mean entropy of attributes and class entropy are examples of information-theoretic features.

Landmarking

Landmarking characterizes the dataset by how simple machine learning algorithms perform on them. Examples include the performance of a decision stump, naive Bayes classifier or linear discriminant analysis [19, 2].

Alternatively, more complex algorithms can be used on a subset of the data. To allow these meta-features to be calculated quickly, a constant-sized training set is used and the performance is measured based on a fraction of the data. These types of landmarks are called sampling-based landmarks [23]. Algorithms used in [23] for sampling-based landmarking include decision trees, neural networks and an ensemble of boosted trees.

According to [19] good landmarkers should probably have a run-time complexity of at most $O(n \log n)$. Otherwise, calculating landmark features could become more expensive than executing the complex learner itself .

Moreover, when using multiple landmarkers, they should have different biases [19, 2]. If two distinct landmarkers have similar performance across all datasets, then it probably suffices to only use one of them [12].

Model-based

Similar to landmarking, model-based features are constructed by running machine learning algorithms on the dataset. However, the features are based on the model constructed by the

learner. For example, the decision tree algorithm can be used to construct model-based meta-features. First, a decision tree is learned from the data. Then, characteristics of this decision tree are the meta-features. Examples include its depth or the number of leaves. Moreover, there are several characteristics of which the minimum, maximum, mean and standard deviation are used, such as the branch-length, the number of nodes per level or the number of times an attribute is used in a split [18].

2.3.2 Warm-starting

Meta-learning is complementary to Bayesian optimization for optimizing an ML framework. Meta-learning can quickly suggest some hyperparameter settings that are likely to perform quite well, but it is unable to provide fine-grained information on performance. In contrast, Bayesian optimization is slow to start for hyperparameter spaces as large as those of entire ML frameworks, but can fine-tune performance over time. We exploit this complementary by selecting k configurations based on meta-learning and use their result to seed Bayesian optimization. This approach is called warm-starting.

2.4 Multi-armed bandit problem

The multi-armed bandit problem is a problem in which a fixed set of resources must be allocated between competing choices in a way that maximizes their expected gain, when the choice's properties are only partly known at the time of allocation, and may become better understood as time passes or by allocating resources to the choice.

The name of the problem comes from a slot machine, which is sometimes known as a one-armed bandit. A slot machine is operated by pulling down a lever at one side. This operating lever looks like an arm, and the machine in effect robs players, since it "wins" and keeps the player's money in an overwhelming majority of instances.

Now imagine a row of slot machines (i.e. a multi-armed bandit) where each machine provides a random reward from a probability distribution specific to that machine. The objective of the gambler is to maximize the sum of rewards earned through a sequence of lever pulls. Many real-world learning and optimization problems can be modeled in this way.

Several algorithms have been proposed for the multi-armed bandit problem [21, 1, 24].

2.4.1 Greedy Strategies

These strategies were the earliest and simplest strategies discovered to approximately solve the bandit problem. These strategies have in common a greedy behavior where the best lever (based on previous observations) is always pulled except when a (uniformly) random action is taken.

Epsilon-greedy

The best lever is selected for a proportion $1 - \varepsilon$ of the trials, and a lever is selected at random (with uniform probability) for a proportion ε.

Epsilon-decreasing

Similar to the epsilon-greedy strategy, except that the value of ε decreases as the experiment progresses, resulting in highly explorative behaviour at the start and highly exploitative behaviour at the finish.

2.4.2 Thompson sampling methods

Suppose there are K levers or actions, and when played, any action yields either a success or a failure. Action $k \in 1, ..., K$ produces a success with probability $0 \leq \theta_k \leq 1$. The success

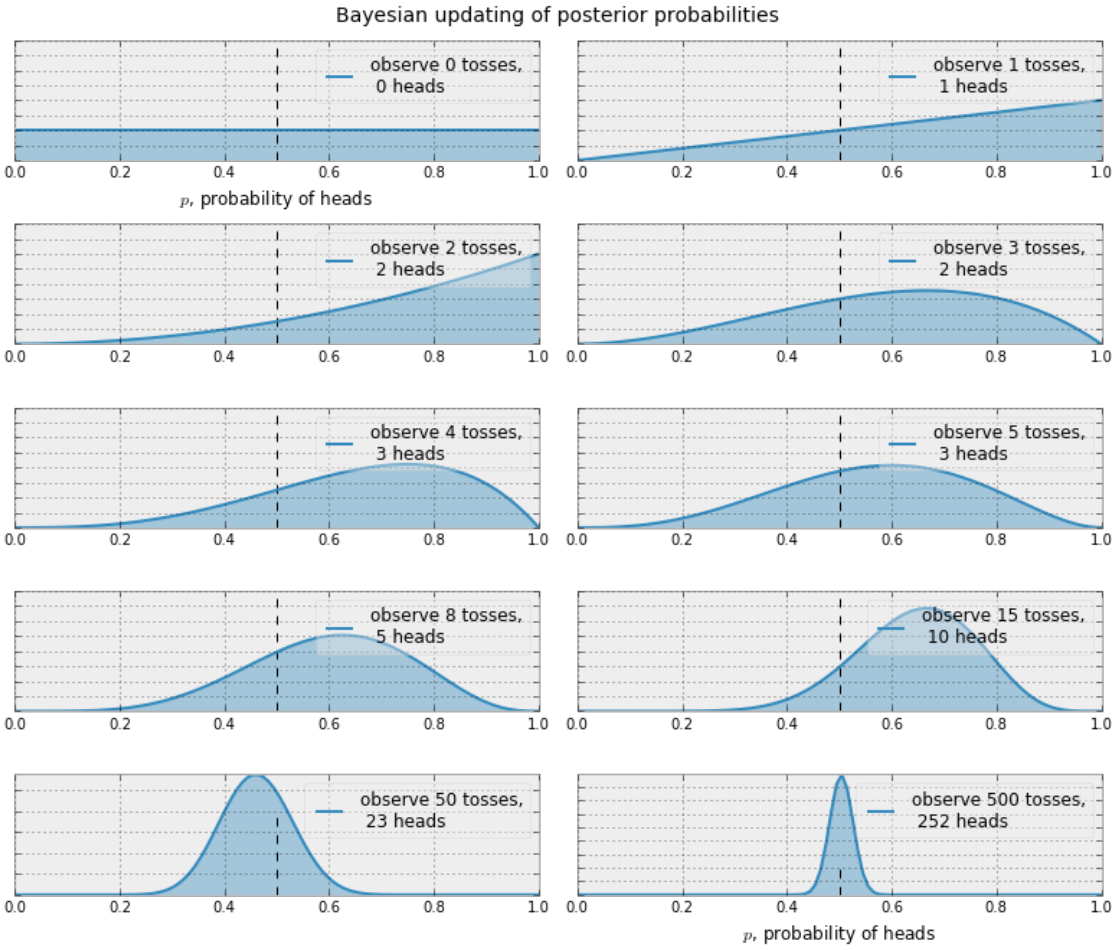


Figure 2.7: Visualization of how the Beta-distribution changes after a few coin flips.

probabilities $(\theta_1, .., \theta_K)$ are unknown to the agent, but are fixed over time, and therefore can be learned by experimentation.

Thompson sampling [24] keeps track of the number of times an action k had success s_k and the number of times the action led to failure f_k. These variables will then be used as the input of the beta-distribution.

The beta distribution is a family of continuous probability distributions defined on the interval $[0, 1]$ parametrized by two positive shape parameters, denoted by α and β, that appear as exponents of the random variable and control the shape of the distribution.

Figure 2.7 [2] shows how the shape of the beta-distribution changes as we observe the results of a coin-flipping experiment. At the start, it takes the form of a uniform distribution, and over time, a pike starts to form around the true probability, marked with the dashed line.

On each iteration, the Thompson sampling method draws values $\hat{\theta}_1, ..., \hat{\theta}_k$ from K beta distributions defined by $\alpha = s_k$ and $\beta = f_k$. The action with the highest value is then chosen, i.e: $\arg\max_k \hat{\theta}_k$. The method is shown in Algorithm 1.

Figure 2.8 visualizes Thompson Sampling with three levers. The bars show the underlying true success-rates of the levers and the corresponding beta-distributions are shown as overlapping graphs. When we sample values from the distributions, the height of these graphs determine the probability of receiving a value at that point. The crosses at the bottom are a possible result of drawing these samples from their distributions. In this case, we received our highest sample

[2] https://wiki.math.uwaterloo.ca/statwiki/index.php?title=File:thompson_sampling_coin_example.png

Algorithm 1 Thompson Sampling

procedure SAMPLEBETA(K, α, β)
  for $k \leftarrow 1, K$ **do**
    Sample $\hat{\theta}_k \sim beta(\alpha_k, \beta_k)$
  end for
  $k \leftarrow \arg\max_k \hat{\theta}_k$
end procedure

procedure THOMPSONSAMPLING(K, α, β)
  $k \leftarrow$ SAMPLEBETA(K, α, β)
  Apply action k and observe reward r
  $\alpha_k \leftarrow \alpha_k + r$
  $\beta_k \leftarrow \beta_k + 1 - r$
end procedure

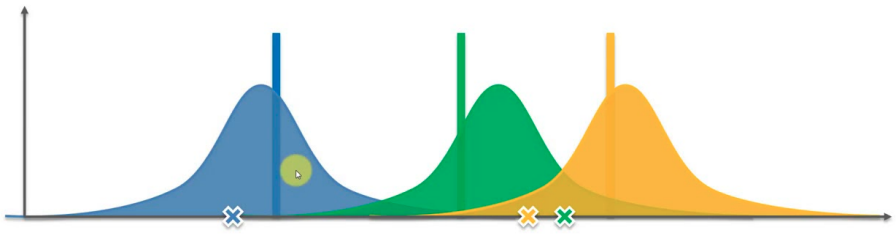


Figure 2.8: Example draw using Thompson Sampling with three levers.

from the green distribution, and so the green lever will be pulled and we observe its reward. The corresponding beta-distribution will then be updated with this new observation.

Changes to Thompson sampling

In our experiments, our actions produce continuous reward between 0 and 1 rather than a binomial variable. To make Thompson sampling applicable to our case, we we can derive the α and β values from the mean (μ) and standard deviation (σ) of our rewards.

We first start with the properties of the beta distribution:

$$\mu = \frac{\alpha}{\alpha + \beta} \tag{2.11}$$

$$\sigma^2 = \frac{\alpha\beta}{(\alpha + \beta)^2 * (\alpha + \beta + 1)} \tag{2.12}$$

Next, we solve for β:

$$(\alpha + \beta)\mu = \alpha \tag{2.13}$$

$$\alpha + \beta = \frac{\alpha}{\mu} \tag{2.14}$$

$$\beta = \alpha \left(\frac{1}{\mu} - 1 \right) \tag{2.15}$$

We introduce a placeholder t, where $t = \frac{1}{\mu} - 1$. And finally, we solve for α:

$$\beta = \alpha t \tag{2.16}$$

$$\sigma^2 = \frac{\alpha^2 t}{(\alpha + \alpha t)^2 * (\alpha + \alpha t + 1)} \tag{2.17}$$

$$\sigma^2 = \frac{\alpha^2 t}{\alpha^2 (1+t)^2 * (\alpha(1+t) + 1)} \tag{2.18}$$

$$\sigma^2 = \frac{t}{(1+t)^2 * (\alpha(1+t) + 1)} \tag{2.19}$$

$$\alpha(1+t) + 1 = \frac{t}{(1+t)^2 * \sigma^2} \tag{2.20}$$

$$\alpha = \frac{\frac{t}{(1+t)^2 * \sigma^2} - 1}{1+t} \tag{2.21}$$

$$\alpha = \frac{\frac{t}{1+t}}{(1+t)^2 \sigma^2} - \frac{1}{1+t} \tag{2.22}$$

Now, note that $\frac{t}{1+t} = \frac{\frac{1}{\mu} - 1}{\frac{1}{\mu}} = \mu - 1$ and so:

$$\alpha = \frac{1-\mu}{\frac{1}{\mu^2}\sigma^2} - \mu \tag{2.23}$$

$$\alpha = \left(\frac{1-\mu}{\sigma^2} - \frac{1}{\mu}\right) \mu^2 \tag{2.24}$$

Note that $\sigma > 0$ is required. A special case occurs when $\sigma_k = 0$. In this case we assume that there is not enough information for action k and simply apply this action to gather more information. See Algorithm 2.

Algorithm 2 Thompson Sampling for continuous rewards

procedure SAMPLEBETA(K, A)
  for $k \leftarrow 1, K$ **do**
    $\mu \leftarrow \text{MEAN}(A_k)$
    $\sigma \leftarrow \text{STANDARDDEVIATION}(A_k)$
    if $\sigma_k = 0$ **then**
      return k
    end if
    $\alpha_k \leftarrow \left(\frac{1-\mu}{\sigma_k^2} - \frac{1}{\mu_k}\right) \mu_k^2$
    $\beta_k \leftarrow \alpha_k \left(\frac{1}{\mu_k} - 1\right)$
    Sample $\hat{\theta}_k \sim beta(\alpha_k, \beta_k)$
  end for
  $k \leftarrow \arg\max_k \hat{\theta}_k$
  return k
end procedure

procedure THOMPSONSAMPLING(K, A)
  $k \leftarrow \text{SAMPLEBETA}(K, A)$
  Apply action k and observe reward r
  $A_k.insert(r)$
end procedure

Top-Two Thompson Sampling

Thompson sampling can have very poor asymptotic performance for the best arm identification problem [21]. Intuitively, this is because once it estimates that a particular arm is the best with reasonably high probability, it selects that arm in almost all periods at the expense of refining its knowledge of other arms.

Russo proposes top-two Thompson sampling (TTTS) [21], which modifies standard Thompson sampling by adding a re-sampling step. This algorithm depends on a tuning parameter $\gamma > 0$ that will sometimes be set to a default value of $\frac{1}{2}$. In comparison, Russo shows that Thompson Sampling requires 60% more measurements to reach confidence .95 and over 250% more measurements to reach confidence .99 of the probability value for the best lever.

We show this algorithm in Algorithm 3. Note that we can easily modify this algorithm to use actions with continuous rewards, in the same fashion as we did in the previous section.

Algorithm 3 Top-Two Thompson Sampling

```
1: procedure TTTS(K, α, β, γ)
2:     k ← SampleBeta(K, α, β)
3:     Sample R ~ Bernoulli(γ)
4:     if R = 1 then
5:         Apply action k and observe reward r
6:     else
7:         repeat
8:             i ← SampleBeta(K, α, β)
9:         until i ≠ k
10:        Apply action k and observe reward r
11:    end if
12:    α_k ← α_k + r
13:    β_k ← β_k + 1 − r
14: end procedure
```

2.4.3 UCB1 and UCB1-tuned

UCB1 [1] uses the principle of "optimism in the face of uncertainty". We believe an action is as good as possible given the available evidence. Our optimism comes in the form of an upper confidence bound. Specifically, we want to know with high probability, that the true expected payoff of an action μ_i is less than our prescribed upper bound.

The UCB1 algorithm uses a formula based on the Chernoff-Hoeffding inequality. The Chernoff-Hoeffding inequality is distribution independent and provides an exponential upper bound on the probability that a random variable X deviates from its mean. Let $X_1, X_2, \ldots, X_n$ be independent random variables in the range $[0, 1]$ with $\mathbb{E}[X_i] = \mu$. Then for upperbound $a > 0$, the following holds:

$$P\left(\frac{1}{n}\sum_{i=1}^{n} x_i \geq \mu + a\right) \leq e^{-2na^2}$$

We can then solve this equation for a to find an upper bound big enough to be confident that we are within a of the true mean. We solve $p \leq e^{-2na^2}$ for a, which gives:

$$a = \sqrt{\frac{\ln p}{-2n}}$$

We then reduce p as we observe more rewards, for example $p = t^{-4}$, where t is the total number of actions we have tried. This ensures we select the optimal action as $t \to \infty$.

$$a = \sqrt{\frac{2 \ln t}{n}}$$

This leads us to the UCB1 algorithm. The algorithm records the average reward $\hat{x}_k$ for each action k and number of times we have tried it: n_k . We write t for total number of actions we have tried. Then we try the action that maximizes

$$\hat{x}_k + \sqrt{\frac{2 \ln t}{n_k}}$$

UCB1-tuned

For practical purposes, the bound of UCB1 can be tuned more finely. UCB1-tuned [1] replaces the upper confidence bound $\sqrt{\frac{2 \ln t}{n_k}}$ of policy UCB 1 with

$$\sqrt{\frac{\ln t}{n_k} \min\left(1/4, V_k(n_k)\right)}$$

where $V_k(s)$ is defined as follows:

$$V_k(s) = \left(\frac{1}{s} \sum_{\tau=1}^{s} \hat{X}_{k,s}^2 + \sqrt{\frac{2 \ln t}{s}} \right)$$

Here, k denotes the lever, which has been played s times during the first t plays, has a variance that is at most the sample variance plus $\sqrt{(2 \ln t)/s}$. The factor $1/4$ is an upperbound on the variance of a Bernoulli random variable.

In the experiments performed in [1] it performs substantially better than UCB1 in essentially all of the experiments.

2.5 TPOT and Autosklearn

TPOT and Autosklearn are two of the most popular AutoML algorithms. We will discuss and compare them, as they both provide a different and complete method of solving the CASH problem.

2.5.1 TPOT

TPOT [17] can design machine learning pipelines that provide a significant improvement over a basic machine learning analysis while requiring little to no input nor prior knowledge from the user. TPOT implements the pipelines as trees with the different operators being nodes in the tree.

Every tree-based pipeline begins with one or more copies of the input data set as the leaves of the tree, which is then fed into one of the four classes of pipeline operators: preprocessing, decomposition, feature selection, or modeling. As the data is passed up the tree, it is modified by that nodes operator. When there are multiple copies of the data set being processed, it is possible to combine them into a single data set via a data set combination operator.

Figure 2.9 shows an example tree-based machine learning pipeline. The data set flows through the pipeline operators, which add, remove, or modify the features in a successive manner. Combination operators allow separate copies of the data set to be combined, which can then be provided to a classifier to make the final classification.

To automatically generate and optimize these tree-based pipelines, TPOT uses an evolutionary computation technique called genetic programming (GP).

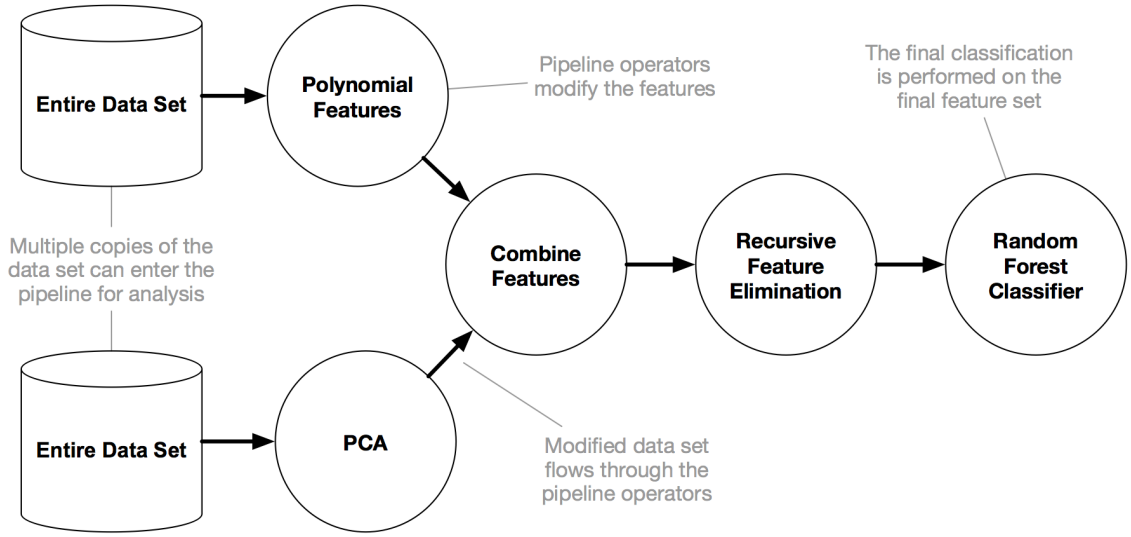


Figure 2.9: An example TPOT pipeline.

2.5.2 SMAC

AutoSklearn [9] is based on SMAC [13] (Sequential Model-based Algorithm Configuration). To understand what AutoSklearn is, we will therefore first discuss SMAC. The SMAC algorithm is an extension of ROAR (Random Online Agressive Racing). ROAR consists of four components: Initialize, FitModel, SelectConfigurations, and Intensify.

- The Initialize component performs a single run with the target algorithm's default parameter configuration.
- The FitModel procedure simply returns a constant model which is never used
- SelectConfigurations returns a single configuration sampled uniformly at random from the parameter space.
- The Intensify component governs how many evaluations to perform with each configuration and when to trust a configuration enough to make it the new best (the incumbent).

SMAC implements the FitModel component as a Random Forest, and the SelectConfigurations component then selects configurations based on the expected improvement (EI) over the best configuration seen so far. SMAC indentifies configurations with a large EI by performing a simple multi-start local search and then considers all resulting configurations with locally maximal EI. However, every second iteration, SMAC skips this multi-start local search and simply evaluates one randomly drawn sample in order to improve the model's interpolation.

Multi-start local search

SMAC's multi-start local search approach first computes the EI for all configurations used in previous target algorithm runs, then picks the ten configurations with maximal EI, and initializes a local search at each of them.

To handle mixed categorical/numerical spaces, they use a randomized one-exchange neighbourhood, including the set of all configurations that differ in the value of exactly one discrete parameter, as well as four random neighbours for each numerical parameter. Since batch model predictions (and thus batch EI computations) are much cheaper than separate predictions, SMAC uses a best improvement search, evaluating EI for all neighbors at once, and stops each local search once none of the neighbours has a larger EI.

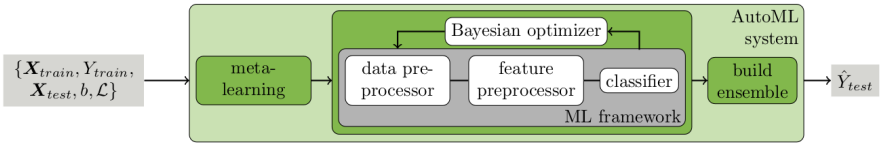


Figure 2.10: AutoSklearn's AutoML approach.

Additionally, SMAC samples an additional 10 000 configurations randomly, and calculates their EI as well. It then sorts these results together with the 10 local-search results. The 10 local-search results typically had larger EI than all randomly sampled configurations according to hutter et al [13]. From this list of 10 010 configurations, SMAC evaluates the configuration with the highest EI.

2.5.3 AutoSklearn

AutoSklearn [9] extends the SMAC algorithm. It includes a meta-learning step in the AutoML pipeline to warm-start the Bayesian optimization procedure, which results in a considerable boost in the efficiency of the system. Second, it includes an automated ensemble construction step, allowing us to use all classifiers that were found by Bayesian optimization. These two additions are shown in Figure 2.10.

Automated ensemble construction

For automated ensemble construction, the authors from Autosklearn paper experimented with stacking [26], gradient-free numerical optimization and ensemble selection [5]. They found both numerical optimization and stacking to overfit to the validation set and to be computationally costly. However, ensemble selection was fast and robust. In a nutshell, ensemble selection (introduced by Caruana et al.) is a greedy procedure that starts from an empty ensemble and then iteratively adds the model that maximizes ensemble validation performance (with uniform weight, but allowing for repetitions).

Meta-learning

AutoSklearn's meta-learning approach works as follows. In an offline phase, for each machine learning dataset in a dataset repository, it evaluated a set of meta-features and used Bayesian optimization to determine and store an instantiation of the given algorithm or pipeline with good performance for that dataset. Then, given a new dataset D, AutoSklearn computes its meta-features, ranks all datasets by their L_1 distance to D in meta-feature space and selects the stored instantiations for the $k = 25$ nearest datasets for evaluation before starting Bayesian optimization with their results.

2.6 Gradient boosting techniques

In our experiments, we mainly use machine learning algorithms from the Scikit-learn framework. We extend our set of machine learning algorithms with popular gradient boosting algorithms, which we will discuss here.

2.6.1 Boosting and bagging

When considering ensemble learning, there are three primary methods: bagging, boosting and stacking. We will discuss bagging and boosting.

- Bagging involves the training of many independent models and combines their predictions through some form of aggregation (averaging, voting etc.). An example of a bagging ensemble is a Random Forest.

- Boosting instead trains models sequentially, where each model learns from the errors of the previous model. Starting with a weak base model, models are trained iteratively, each adding to the prediction of the previous model to produce a strong overall prediction.

In the case of gradient boosted decision trees, successive models are found by applying gradient descent in the direction of the average gradient, calculated with respect to the error residuals of the loss function, of the leaf nodes of previous models.

2.6.2 Boosting types

Different boosting types have been proposed, with GBDT being the most popular [10, 11, 20, 15]. In general, boosting types differ in the way they control over-fitting (e.g. regularization) and how they implement stochastic gradient descent (e.g. sub-sampling).

GBDT

Gradient Boosting Decision Trees (GBDT), also known as Multiple Additive Regression Trees (MART) [10, 11], is an ensemble model of boosted regression trees. It produces a prediction model in the form of an ensemble of decision trees. It builds the model in a stage-wise fashion, and it generalizes them by allowing optimization of an arbitrary differentiable loss function.

However, it suffers an issue wherein trees added at later iterations tend to impact the prediction of only a few instances, and make negligible contribution towards the remaining instances. This negatively affects the performance of the model on unseen data, and also makes the model over-sensitive to the contributions of the few, initially added trees.

Rashmi et al. [20] show that the commonly used tool to address this issue, that of shrinkage, alleviates the problem only to a certain extent and the fundamental issue of over-specialization still remains.

DART

Dropouts meet Multiple Additive Regression Trees (DART) was proposed by Rashmi and Gilad-Bachrach [20] to add dropout techniques from the deep neural net community to boosted trees, and reported better results in some situations. DART diverges from MART at two places:

- When computing the gradient that the next tree will fit, only a random subset of the existing ensemble is considered.

- When adding the new tree to the ensemble, DART re-scales the new tree T by a factor of $1/k$, where k is the number of dropped trees. In this manner, the new tree will have the same order of magnitude as the dropped trees. Following this, the new tree and the dropped trees are scaled by a factor of $\frac{k}{k+1}$, to ensure that the combined effect of the dropped trees together with the new tree remains the same as the effect of the dropped trees. Finally, the new tree is added to the ensemble.

Compared to GBDT, the disadvantages for DART are:

- Training can be slower than GBDT because the random dropout prevents usage of a prediction buffer. A prediction buffer is a buffer that saves the prediction results of last boosting step.

- The early stop might not be stable, due to the random dropouts.

GOSS

Gradient-based One-Side Sampling (GOSS) [15] excludes a significant proportion of the data instances with small gradients, and only uses the rest to estimate the information gain. According to the definition of information gain, those instances with larger gradients (i.e., under-trained instances) will contribute more to information gain.

In order to retain the accuracy of information gain estimation, GOSS keeps the instances with large gradients (e.g., larger than a pre-defined threshold, or among the top percentiles), and only randomly drop those instances with small gradients. Chen et al. [15] claim that such a treatment can lead to a more accurate gain estimation than uniformly random sampling, especially when the value of information gain has a large range.

2.6.3 Gradient boosting frameworks

Since the first implementation in scikit-learn (2012) and XGBoost (2014), a number of new Python libraries have emerged: h2o (2015), LightGBM (2016) and CatBoost (2017). We will briefly cover Scikit-learn and XGBoost and explain LightGBM and Catboost in a bit more detail.

Scikit-learn and XGBoost

Gradient-boosting builds the model in a stage-wise fashion, as it needs to calculate the gradient after adding each tree. Unsurprisingly, the first implementation in scikit-learn is therefore not parallelized and utilizes only one core.

Later algorithms such as XGBoost [6] make gradient boosting parallel, not by creating multiple decision trees in parallel, but using parallelization within a single tree. These algorithms collect statistics for each column in parallel, which leads to a parallel algorithm for split finding. Further optimizations for XGBoost deal with sparsity and cache-awareness.

LightGBM

LightGBM introduced GOSS (see section 2.6.2) and Exclusive Feature Bundling (EFB). EFB bundles features together which are (almost) exclusive, i.e., they rarely take nonzero values simultaneously (e.g. one-hot-features). Bundling these features together makes the feature space less sparse. LightGBM proposed an efficient algorithm for EFB by reducing the optimal bundling problem to a graph coloring problem (by taking features as vertices and adding edges for every two features if they are not mutually exclusive), and solving it by a greedy algorithm with a constant approximation ratio [15].

The experiments show that LightGBM can accelerate the training process by up to over 20 times while achieving almost the same accuracy, compared to Stochastic Gradient Boosting (SGB) without Exclusive Feature Bundling. The experiments furthermore show that LightGBM can significantly outperform XGBoost and SGB in terms of computational speed and memory consumption.

CatBoost

CatBoost requires the user to specify which features are numerical and which are categorical. CatBoost takes the input (e.g. Figure 2.11) and generates randomly shuffled copies of this input (e.g. Figure 2.12. For each copy, CatBoost then calculates a numerical statistic for each category and replaces the categorical values with these statistics (see Figure 2.13).

In regression problems, binarization is first performed on the target value. The mode and number of bins are set by the user. For classification and regression (after binarization), CatBoost calculates the following statistic in a linear fashion:

$$\texttt{avg\_target} = \frac{\texttt{countInClass} + \texttt{prior}}{\texttt{totalCount}}$$

Where:

Object #	f_1	f_2	...	f_n	Function value
1	2	40	...	rock	1
2	3	55	...	indie	0
3	5	34	...	pop	1
4	2	45	...	rock	0
5	4	53	...	rock	0
6	2	48	...	indie	1
7	5	42	...	rock	1
...					

Figure 2.11: CatBoost accepts a set of object properties and model values as input.

- `countInClass` is how many times the label value was equal to 1 for objects with the current categorical feature value.
- `prior` is the preliminary value for the numerator. The value is determined by the user. A default value of 0.05 is used.
- `totalCount` is the total number of objects (up to the current one) that have a categorical feature value matching the current one.

Finally, CatBoost uses different permutations for training distinct models, thus using several permutations does not lead to overfitting.

CatBoost uses oblivious decision trees as base learners. Such trees are balanced and less prone to overfitting. Figure 2.14 shows such a decision tree in comparison with a regular decision tree. In oblivious trees each leaf index can be encoded as a binary vector with length equal to the depth of the tree. This fact is used to speed up model training and prediction.

Object #	f_1	f_2	...	f_n	Function value
1	4	53	...	rock	0
2	3	55	...	indie	0
3	2	40	...	rock	1
4	5	42	...	rock	1
5	5	34	...	pop	1
6	2	48	...	indie	1
7	2	45	...	rock	0
...					

Figure 2.12: The rows in the input file are randomly shuffled several times. Multiple random permutations are generated.

Object #	f_1	f_2	...	f_n	Function value
1	4	53	...	0,05	0
2	3	55	...	0,05	0
3	2	40	...	0,025	1
4	5	42	...	0,35	1
5	5	34	...	0,05	1
6	2	48	...	0,025	1
7	2	45	...	0,5125	0
...					

Figure 2.13: All categorical feature values are transformed to numerical.

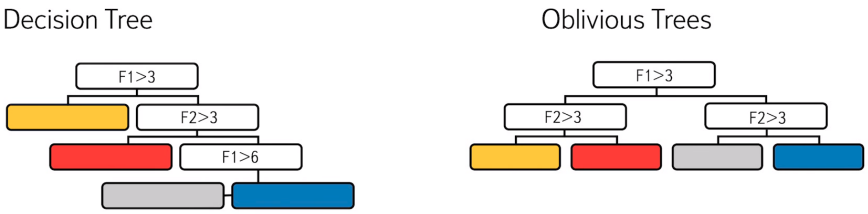


Figure 2.14: Regular versus oblivious decision trees.

Chapter 3

Preparation

In this chapter we will discuss two problems that need to be overcome before we can continue with Bayesian Optimization methods and meta-learning.

The first problem is a preprocessing problem. Due to the dynamic nature of variables in Python, our machine learning framework Scikit-learn includes parameters that can accept a variety of different value types. On the other hand, our estimators expect a clear structure in their features. In section 3.1 we will explain our method of transforming these values into a usable feature set.

The second problem is about quantifying predictive uncertainty for tree-based models. Where Gaussian Processes can calculate the standard deviation of predictive distribution at query points to represent uncertainty, standard Random Forests do not have a way to quantify predictive uncertainty. However, to calculate the Expected Improvement, we need a value for uncertainty. In section 3.2 we will explain different solutions proposed to address this problem.

3.1 Preprocessing

The set of parameters of Scikit-learn algorithms are typically a mix of numerical and nominal (categorical) parameters, and because of the dynamic nature of variables in Python, a parameter can accept both numerical and nominal data. Additionally, Scikit-learn often distinguishes between integers and floating point numbers: integers are absolute numbers, while a floating point number ($0 \leq p \leq 1$) is interpreted as being a fraction of the dataset.

Next to these types, we use a special parameter "@preprocessor" in this thesis that determines which preprocessor to include in our pipeline. This parameter includes several object instances.

The regression techniques from Scikit-learn can only handle numerical data. So, in order to estimate the performance of a given hyperparameter setting, we need to convert any setting to an array of numerical data.

3.1.1 Hyperparameter transformation

In this subsection we will cover how we transform a set of algorithm settings into a set of features, such that it can be used to train an estimator. Figure 3.1 shows a global overview of our preprocessing pipeline, of which we will explain its steps in more detail here.

Type detection

We use a simple type detection system to distinguish between numeric, nominal, boolean, and mixed parameters. Additionally, we detect parameters that accept other objects, such as arrays or mappings.

Ignoring None-types, this system first detects whether the parameter domain contains any non-primitive instances (strings, numbers, booleans), in which case we will label it as "object". If

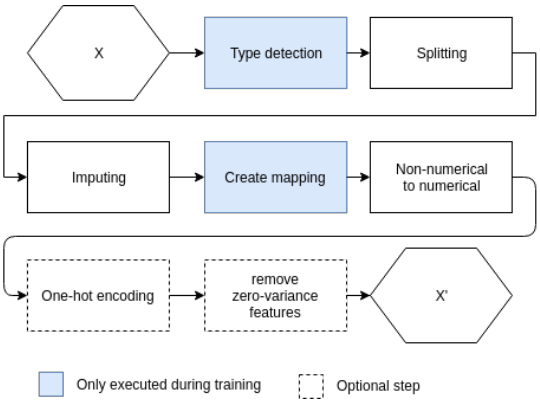


Figure 3.1: Preprocessing steps for hyperparameters.

this is not the case, then it checks if there are any booleans in the domain, in which case it will be labeled as "boolean".

In the case that all values are numerical, we will label this one as "numeric". Or, in the case that all values are non-numerical, we will label this parameter as "nominal". All parameters that do not belong to any category so far, will be labeled as "mixed".

parameter	**unique values**	**type**
n_estimators	10, 100, 500	numerical
criterion	"gini", "entropy"	nominal
max_features	0.05, 0.10, "log2", "sqrt", None	mixed
max_depth	9, 10, 11, 12, None	mixed
min_samples_split	2, 3, 5, 8	numerical
min_samples_leaf	1, 2, 7, 9	numerical
bootstrap	True, False, 0, 1	boolean
@preprocessor	OneHotEncoder(), PCA(), None	object

Table 3.1: Type detection.

Splitting mixed parameters

For a hyperparameter column that contains both numeric and nominal values, we can split this column in a column that contains only nominal values and a column that contains only numeric values. We then treat the unknown values as missing values. Table 3.2 shows the result of splitting the `max_features` parameter of Scikit-learn's Random Forest. Note that `max_features` takes integer values as absolute values and float values as a fraction of the number of features. We could make two different columns for that, however in our case we have only floating point values in our meta-dataset, so there is no need to do so.

max_features	**max_features_nominal**	**max_features_numerical**
"log2"	"log2"	null
None	"None"	null
0.10	null	0.10
"sqrt"	"sqrt"	null
0.20	null	0.20
0.30	null	0.30

Table 3.2: Splitting a hyperparameter with a mixed domain.

In the resulting nominal column, we simply treat the None-types as another category. For the numerical column, we can fill these missing values using a form of imputation, or compute their actual values using the nominal column, if we have specific knowledge about the ML-framework and the dataset on which we run the algorithm. We use the latter method, which we will describe next.

parameter	**values**	**type**
n_estimators	10, 100, 100, 500, 100	numerical
criterion	“gini”, “gini”, “gini”, “entropy”, “gini”	nominal
max_features_num	0.05, 0.10, null, null, null	numerical
max_features_nom	”null”, ”null”, “log2”, “sqrt”, “None”	nominal
max_depth_num	9, 10, 11, 12, null	numerical
max_depth_nom	”null”, ”null”, ”null”, ”null”, “None”	nominal
min_samples_split	2, 3, 8, 5, 8	numerical
min_samples_leaf	1, 7, 2, 7, 9	numerical
bootstrap	True, False, 0, 1, True	boolean
@preprocessor	OneHotEncoder(), PCA(), PCA(), PCA(), None	object

Table 3.3: Parameter values and types after splitting.

Parameter specific imputing

- The hyperparameter `max_features` accepts an integer (absolute number) or a float (percentage of number of features `n_features`) or the values “auto”, “sqrt”, “log2” or null. The values “auto” and “sqrt” both represent $\sqrt{\texttt{n\_features}}$, while “log2” represents $\ln(\texttt{n\_features})$ and null represents no maximum number of features, i.e. `n_features`. The numerical values are all all converted to float percentages by applying the values to the number of features. The missing numerical values are computed from the nominal values. Additionally, because “auto” and “sqrt” mean the same thing, “auto” is renamed to “sqrt” in the nominal column.

- The hyperparameter `gamma` used by the SVM classifier can be a float, or auto. For “auto”, $1/\texttt{n\_features}$ will be used in the numerical column.

- Similar preprocessing is done with `min_samples_leaf` and `min_samples_split` which are based on the number of samples in the dataset.

- If no maximum depth is set for the `max_depth` parameter, it is hard to compute a numerical value. We could calculate an upper bound depending on the number of minimum samples required for a leaf, the minimum number of samples for a split, the minimum weight fraction, and the maximum number of leaf nodes. Instead, we use a constant value $c = 20$ that we believe to be sufficiently high (i.e. a higher maximum depth would not lead to much difference in performance), but not too high (so that the effect of a higher maximum depth is not exaggerated).

In our case, we have now resolved all “mixed” parameters. Table 3.4 shows the result.

Creating a mapping

In some cases we would like to create our transforming model first on known data and use it again to transform unknown data. In order to do so, we create a mapping for nominal features from their string representation to their numerical representation, which we will discuss in the next step. Additionally, we add a token for unknown categories that were not seen in the data. Table 3.5 shows an example of such a mapping for a Random Forest.

parameter	values	type
n_estimators	10, 100, 100, 500, 100	numerical
criterion	“gini”, “gini”, “gini”, “entropy”, “gini”	nominal
max_features_num	0.05, 0.10, 0.0576, 0.0913, 1	numerical
max_features_nom	”null”, ”null”, “log2”, “sqrt”, “None”	nominal
max_depth_num	9, 10, 11, 12, 20	numerical
max_depth_nom	”null”, ”null”, ”null”, ”null”, “None”	nominal
min_samples_split	2, 3, 8, 5, 8	numerical
min_samples_leaf	1, 7, 2, 7, 9	numerical
bootstrap	True, False, 0, 1, True	boolean
@preprocessor	OneHotEncoder(), PCA(), PCA(), PCA(), None	object

Table 3.4: Parameter values after imputing.

criterion		max_features_nominal	
key	value	key	value
“<unkn>”	0	“<unkn>”	0
“entropy”	1	“log2”	1
“gini”	2	“None”	2
		“sqrt”	3

Table 3.5: A mapping from nominal features to their string representation.

Converting non-numerical parameters to numerical features

Parameters marked as “object” can contain arrays, collections or class instances. In the general case, we can simply treat these parameters as nominal values, where each unique array, collection or class instance is a new category.

In our case, parameters that are marked as “object” are no active parameters. One example is the one-hot encoder's parameter `categorical_features`, which requires a list of indices or booleans that indicate if a certain feature is categorical or not. Another example is the support vector machine algorithm's parameter `class_weight` which allows for setting a mapping that sets the class weight. We therefore exclude these parameters.

At this point, we are left with nominal, boolean and numerical parameters, or features. We convert the nominal values to numbers using the unique values as indices. For the boolean values, we convert True to 1 and False to 0. Additionally, we keep a list that indicates which features are nominal. Table 3.6 shows the result of this step.

parameter	values	type	nominal
n_estimators	10, 100, 100, 500, 100	numerical	False
criterion	1, 1, 1, 0, 1	nominal	True
max_features_num	0.05, 0.10, 0.0576, 0.0913, 1	numerical	False
max_features_nom	2, 2, 0, 3, 1	nominal	True
max_depth_num	9, 10, 11, 12, 20	numerical	False
max_depth_nom	1, 1, 1, 1, 0	nominal	True
min_samples_split	2, 3, 8, 5, 8	numerical	False
min_samples_leaf	1, 7, 2, 7, 9	numerical	False
bootstrap	1, 0, 0, 1, 1	numerical	False
@preprocessor	1, 2, 2, 2, 0	nominal	True

Table 3.6: Numerical parameter values.

One-hot encoding

As a next step, we can encode the nominal values using one-hot encoding. We can use the mapping we created earlier to post-fix our parameter names with the category name. Table 3.7 shows the result of this step.

We treat this as an optional step. Catboost would namely benefit more if we kept the nominal parameters as we did in Table 3.6, so that it can calculate statistics over the nominal parameters using the indicator that determines whether a parameter is a nominal or numerical feature.

Scikit-learn algorithms on the other hand would benefit from the one-hot-encoding. The one-hot encoding avoids that these algorithms wrongly interpret nominal feature values as having a numeric relationship (i.e. a category labeled "2" is not higher than a category labeled "1"). Furthermore, LightGBM can use Exclusive Feature Bundling to bundle our one-hot encoded features together so that the training and prediction times decrease.

parameter	**values**
n_estimator	10, 100, 100, 500, 100
criterion_gini	1, 1, 1, 0, 1
criterion_entropy	0, 0, 0, 1, 0
max_features_num	0.05, 0.10, 0.0576, 0.0913, 1
max_features_nom_null	1, 1, 0, 0, 0
max_features_nom_None	0, 0, 0, 0, 1
max_features_nom_sqrt	0, 0, 0, 1, 0
max_features_nom_log2	0, 0, 1, 0, 0
max_detph_num	9, 10, 11, 12, 20
max_depth_nom_null	1, 1, 1, 1, 0
max_depth_nom_None	0, 0, 0, 0, 1
min_samples_split	2, 3, 8, 5, 8
min_samples_leaf	1, 7, 2, 7, 9
bootstrap	1, 0, 0, 1, 1
@preprocessor_OneHotEncoder	1, 0, 0, 0, 0
@preprocessor_PCA	0, 1, 1, 1, 0
@preprocessor_None	0, 0, 0, 0, 1

Table 3.7: Parameter values after one-hot encoding.

Removing zero-variance features

Zero-variance features are features that always keep the same value in every setting or sample. We can remove these features, because they do not contribute to the learning process of an estimator.

3.1.2 Parameter spaces

A parameter space is a user-defined mapping where each parameter comes with a set of options. Our goal is then to find the set of best options. An example parameter space is given in Table 3.8.

We can use the same preprocessing steps we used as shown in Figure 3.1 to convert this parameter space for use in Bayesian Optimization, by using the parameter space as training data for the transformer to fit on. Then, when we sample a set of parameter values from the original parameter space, we can convert these features with our trained transformer.

Additionally, it is possible to define the parameter spaces of our numerical parameters as a distribution using the Scipy[1] package. Scikit-learn's parameter sampler[2] can then draw random

[1] https://docs.scipy.org/doc/scipy-1.1.0/reference/stats.html
[2] http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.ParameterSampler.html

name	values
n_estimators	2^3, 2^4, ..., 2^9
criterion	gini, entropy
max_features	0.05, 0.10, ..., 0.50
max_depth	8, 9, 10, 11, 12, None
min_samples_split	2, 3, ..., 21
min_samples_leaf	1, 2, ..., 21
bootstrap	True, False
@preprocessor	OneHotEncoder(), PCA(), None

Table 3.8: Parameter space of the model

continuous numbers from this distribution. When we define such a distribution in our parameter space, we treat this parameter simply as numerical. In this way, we do not need to do additional steps to transform any of the input values for this parameter. We implement this method as an extension of the type detection step.

3.2 Modifications to tree-based models

Where Gaussian Processes can calculate the standard deviation of predictive distribution at query points to represent uncertainty, standard Random Forests do not have a way to quantify predictive uncertainty. However, to calculate the Expected Improvement, we need a value for uncertainty. In other words, next to a point estimate, the EI needs a standard deviation (which represents uncertainty) to be reported by the estimator.

In 2011, Hutter et al. [13] used a simple method to quantify the predictive uncertainty in random forests, which we will name "tree-level uncertainty". Later in 2014, Hutter et al. [14] published another paper, in which they introduced a new method. We will call this method "leaf-level uncertainty". The latter method is now used in SMAC[3].

3.2.1 Tree-level uncertainty

The random forests predictive mean μ_θ and variance σ_θ^2 for a new configuration θ are computed as the empirical mean and variance of its individual trees predictions for θ.

3.2.2 Leaf-level uncertainty

This method stores, for each leaf of the regression tree, the empirical mean and empirical variance of the training data associated with that leaf.

Some of the leaves may get a low variance when they contain very little data. For example, a leaf with only one sample will have a variance of 0. To avoid that a high certainty will be reported, [14] suggests to round these variances up to at least 0.01.

Since each input for a regression tree will guide you to a different leaf, we can see that for any input, each regression tree T_b now yields a predictive mean μ_b and a predictive variance σ_b^2.

To combine these estimates into a single estimate, we treat the forest as a mixture model of B different models. We can then compute μ and σ^2 as follows:

[3]https://github.com/automl/SMAC3

$$mu = \frac{1}{B} \sum_{b=1}^{B} \mu_b \tag{3.1}$$

$$sigma^2 = \frac{1}{B} \left(\sum_{b=1}^{B} \sigma_b^2 + \mu_b^2 \right) - \mu^2 \tag{3.2}$$

More intuitively, the mean prediction is calculated as the mean of the means and the variance estimate is calculated as the mean of the variance plus the variance of the means.

3.2.3 SMAC's implementation of random forests

AutoSklearn is based on SMAC and therefore inherits some of SMAC's implementation details. AutoSklearn uses the model from SMAC, which is a model that is based on random forests [13].

SMAC constructs a random forest as a set of B regression trees, each of which is built on n data points randomly sampled with repetitions from the entire training data set $\{(\theta_1, o_1), \ldots, (\theta_n, o_n)\}$ where θ_i is a parameter configuration and o_i is the target algorithms observed performance when run with configuration θ_i.

At each split point of each tree, a random subset of $\lceil d \times p \rceil$ of the d algorithm parameters is considered eligible to be split upon. The split ratio p is a parameter, which is set to a default of $p = 5/6$. A further parameter is n_{min}, the minimal number of data points required to be in a node if it is to be split further; we use the standard value $n_{min} = 10$. Furthermore, SMAC sets the number of trees to $B = 10$ to keep the computational overhead small. Finally, SMAC uses the leaf-level uncertainty method as described in the previous subsection.

Chapter 4

Exploration and early experiments

In this chapter, we describe a small part of our early experiments of Bayesian Optimization and discuss the main take-aways from this exploratory phase. This chapter is mainly important to see the differences in performance and running time of Gaussian Processes, Randomized Search and tree-based models.

4.1 Experiment setup

For the experiments we use Lisa Cluster from Surfsara. In particular, we use the nodes with a E5-2650 v2 2.60 GHz processor. Each experiment is allowed to run for 25 minutes, with a maximum of 2 minutes per evaluation. We run each experiment 3 times for each dataset, which gives a total of 12.5 hours (per CPU core). Additionally, each run is started with a randomized search over 5 samples. We use three different seeds for the random number generator, such that each experiment starts with the same 3 sets of 5 samples. The results are gathered and stored in a document database (MongoDB).

4.1.1 Inner and outer loop

We run our experiments in the so-called "inner-loop" mode. Here, the data and the cross-fold-validation folds are downloaded from OpenML. We then use the entire dataset to evaluate the model we try to optimize with cross-fold-validation using the downloaded folds.

This contrasts with the "outer-loop" mode. In the "outer-loop" mode, OpenML will use its cross-fold-validation method to evaluate the performance of our Bayesian Optimization algorithm instead of the performance of the optimized model. This means that we run our Bayesian Optimization once for each of the folds, using the training data for the inner-loop to optimize our model on using our own cross-fold-validation. Once we have optimized the model, we then retrain the optimized model on the full training set, and predict on the test set.

4.1.2 Surrogate models

We use SMAC's implementation of random forests with their default settings. Next to that, we add an Extra-Random Forest (ERF) model with 100 estimators and a maximum of 64 leaves.

Furthermore, we add a Gaussian Process Regressor (GPR). We are using the Matern kernel with a length scale of 1, a lower bound on that length scale of 0.01 and an upperbound of 100. We are using $\nu = 2.5$, which means that the kernel only uses functions that are twice differentiable (i.e., the function is differentiable, and the derivative is also differentiable). The Matern kernel is multiplied by a constant kernel with a constant value of 1 and a lower bound of 0.01, and an upper bound of 1000. We normalize the GP's target values so that the mean of the observed target values become zero. And we set the number of restarts of the optimizer to 2.

For predicting the running time we will use the LightGBM model with 10 estimators, 4 leaves, a learning rate of 0.2 and GBDT boosting.

We compare these results against Randomized Search and the sped-up version Randomized Search x2. We simulate the sped-up version by letting it run twice as long, and then dividing the resulting running times by 2.

4.1.3 Time limit and sampling

We want to be able to abort a setting if its evaluation takes too long. However, whenever a setting runs out of time, this means we will have no measure of its performance. When a setting runs out of time, we insert a score of 0 for that value, to avoid that this sample will be chosen again by the optimizer.

Every iteration, we randomly sample 500 hyperparameters from our parameter space. This parameter space is described in section 4.3.

4.1.4 Post-processing

When all data is gathered, we apply additional post-processing. First we calculate the maximum value we got for each task, and divide the scores we found during optimization by this value. Then we split all our samples into equal-sized time-intervals. Whenever we have an empty bucket, we use the sample from the previous full bucket. Then we take the maximum value found within that bucket, and store the upper-bound of the time-interval. I.e. now we have a list of the best score $f(t)$ we found so far at time t. Finally, for each method, per time-interval, we take the mean of our score over each task.

4.2 Tasks

Table 4.1 shows the tasks that we will run our algorithm on. These tasks were chosen from OpenML , a curated benchmark suite.

task	dataset name	# samples	# features	# classes
12	mfeat-factors	2000	217	10
14	mfeat-fourier	2000	77	10
16	mfeat-karhunen	2000	65	10
20	mfeat-pixel	2000	241	10
22	mfeat-zernike	2000	48	10
28	optdigits	5620	65	10
32	pendigits	10992	17	10
41	soybean	683	36	19
45	splice	3190	62	3
58	waveform-5000	5000	41	3

Table 4.1: Overview of the datasets used for experimentation.

The datasets for 12, 14, 16, 20 and 22 are a set of datasets describing features of handwritten numerals (0 - 9) extracted from a collection of Dutch utility maps. Furthermore, dataset “optdigits” extracts normalized bitmaps of handwritten digits from a preprinted form and “pendigits” is created by recording tablet coordinates and pressure level values of the pen at fixed time intervals.

Dataset “soybean” is for detecting a set of 19 soybean diseases. The goal of “splice” is to recognize, given a sequence of DNA, the boundaries between exons (the parts of the DNA sequence retained after splicing) and introns (the parts of the DNA sequence that are spliced out). “Waveform-5000” generates three classes of waves from a combination of 2 of 3 “base” waves.

4.3 Target-model

The model we use for the experiments is Scikit-learn's random forest classifier. Table 4.2 shows the parameter space we will be using for the experiments. We use an additional parameter `@preprocessor` that determines which preprocessing step will be added to the pipeline. The one-hot-encoder is automatically added or removed from the options, depending on whether the data contains nominal values. Next to that, we use an imputer to impute missing values if there are any.

n_estimators	2^3, 2^4, ..., 2^9
criterion	gini, entropy
max_features	0.05, 0.10, ..., 0.50
max_depth	8, 9, 10, 11, 12, None
min_samples_split	2, 3, ..., 21
min_samples_leaf	1, 2, ..., 21
bootstrap	True, False
@preprocessor	OneHotEncoder(), PCA(), None

Table 4.2: Parameter space of the model

4.4 Value of ξ

Lizotte [4] suggested adding a value ξ to the EI function:

$$EI(x) = (\mu - f^* - \xi)\Phi\left(\frac{\mu - f^* - \xi}{\sigma}\right) + \sigma\phi\left(\frac{\mu - f^* - \xi}{\sigma}\right) \tag{4.1}$$

He then showed that setting $\xi = 0.01$ works well in under certain conditions [16].

To find out if such a parameter works for us, we will run a benchmark over the 10 data-sets mentioned above, with a time-limit of 1500 seconds for each data-set.

Figure 4.1 shows the result of this benchmark. The higher we set our value for ξ, the lower the performance seems to be. Hence, we will stick with the default EI function where $\xi = 0$.

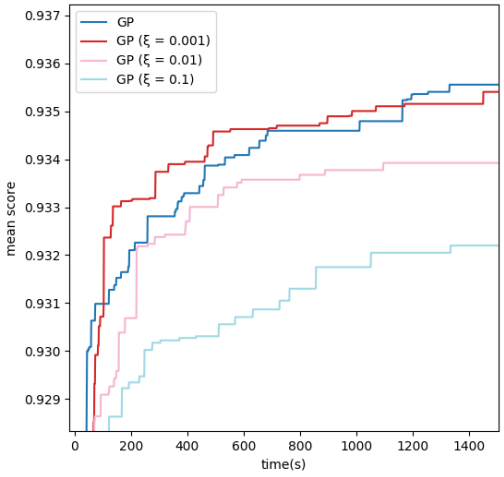


Figure 4.1: Effect of adding a parameter $\xi > 0$ as suggested by [16].

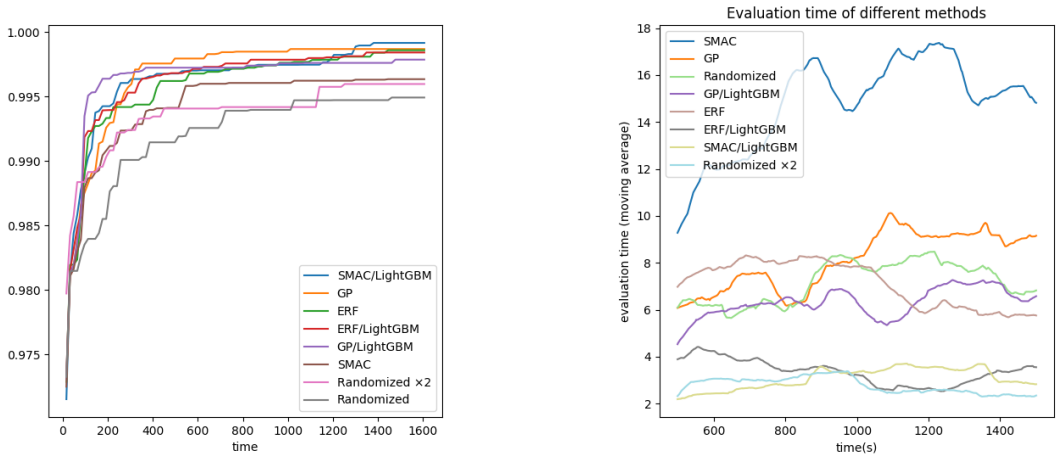
.

4.5 Results

While GP performs quite well, especially in the beginning, Figure 4.3 shows that the time required to find the setting with the highest EI rises much faster than tree-based methods as we do more iterations.

The tree-based models have a speed similar to randomized. However, since we penalize settings that take longer than 2 minutes to evaluate by inserting a score of 0, the Bayesian optimizer will think that this is an area with low performance, and avoid it. This makes it slightly faster than randomized search.

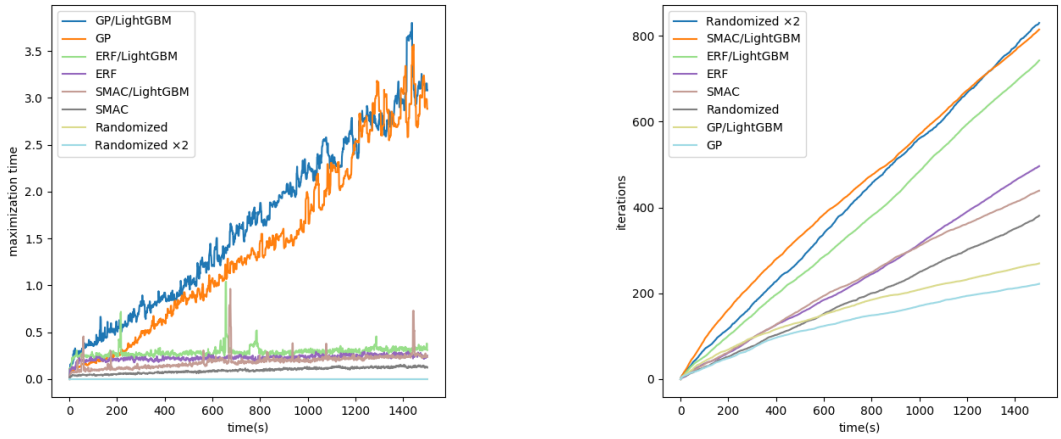
Using EI/s with LightGBM as our time-estimator increases the speed significantly by decreasing the average evaluation time as shown in Figure 4.3.



(a) Method performance

(b) Method evaluation time

Figure 4.2: Score and evaluation time per method.



(a) Time taken to find the value with the highest EI

(b) Number of iterations per second

Figure 4.3: Maximization time and iterations per method.

Chapter 5

Improving hyperparameter optimization with gradient boosting

In this chapter we will propose two new methods for hyperparameter optimization that has shown improvement upon the state-of-the-art performance. In section 5.1 we set our objectives of (1) improving estimation of evaluation cost and (2) improving the quantification of promising configurations. We then propose a method for improving the estimation of evaluation cost in section 5.2, and we propose an improvement on quantifying promising configurations in section 5.3. These two improvements are then combined in section 5.4 and compared on a larger set of datasets.

5.1 Objectives

Our goal is to improve upon the current state-of-the-art AutoML tool, AutoSklearn. Since the hyperparameter optimization itself in AutoSklearn is handled by SMAC, the objective is to improve upon SMAC.

SMAC uses Bayesian Optimization with a Random Forest algorithm as a surrogate model s_p for modelling the target model's performance. Evaluating the target model's performance for a configuration θ is assumed to be expensive, while the surrogate model can evaluate θ cheaply by predicting its performance. SMAC furthermore uses a surrogate model s_r (another instance of the Random Forest algorithm) to model the runtime duration of the target model.

SMAC computes the Expected Improvement (EI) based on s_p's predictive distribution for θ, as described in subsection 2.2.5. And, taking the evaluation cost into account, SMAC computes the runtime duration using s_r to estimate the expected runtime duration for θ. SMAC then combines the expected improvement and evaluation cost by computing the expected improvement per second (EI/s).

We will replace these two surrogate (s_p and s_r) models using a gradient boosting method. Our objectives are then as follows:

- **Evaluation cost:** We desire strong anytime performance on estimating algorithm runtime duration, as we gather more information during the Bayesian Optimization process.

- **Quantify promising configurations:** We want a model that not only estimates the performance of a configuration well, but also expresses its uncertainty about this estimate. Ideally, the model should be able to quantify promising configurations in such a way that by alternately selecting and evaluating configurations, a better configuration can be found faster.

5.2 Evaluation cost

In this section we will determine a model that improves upon the Random Forest used by SMAC in estimating the evaluation cost of a set of configurations. This evaluation cost is defined as the (logarithmic) time it takes to evaluate a configuration.

In Bayesian Optimization we gradually gather more information while optimizing our hyperparameters. Because we are given only limited time to gather this information, the amount of data is limited. We are therefore interested in an estimator performs well with a small amount of data, but still performs well with a larger amount of data, i.e. we want strong anytime performance. We also want to keep the overhead of estimating this evaluation cost low, so that we can invest more time in evaluating configurations.

5.2.1 Choice of model

We will choose LightGBM, a gradient boosting machine, as our surrogate model. LightGBM shows much faster execution time than any Scikit-learn algorithm in a baseline comparison (except KNN, linear regression and decision tree), as well as CatBoost and XGBoost. See Table 5.1. We used data of 40.000 runs on the model described in section 4.3, with different hyperparameter configurations across 10 different datasets as shown in Table 4.1.

We used 10-fold cross-validation on this data, and while validating the performance of our model, we measured the overhead as the average duration to complete one fold (i.e. training and prediction). We computed the average root-mean-squared-error (RMSE) of the predictions.

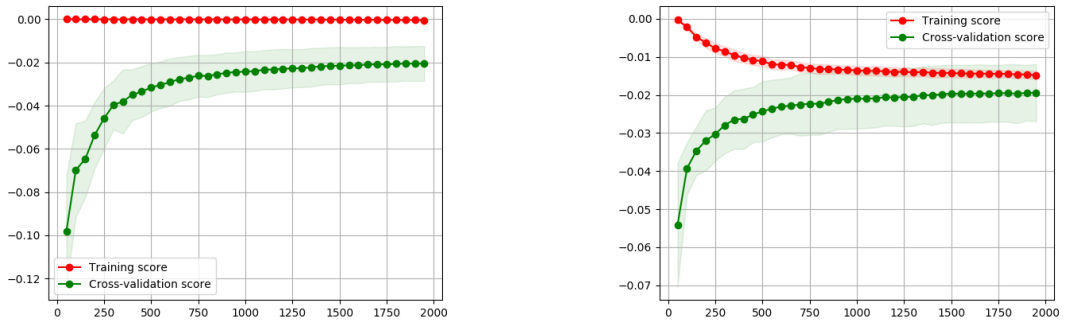
Regressor	**RMSE**	**Overhead (s)**
LGBMRegressor	0.1631	0.0653
CatBoostRegressor	0.164	0.2476
GradientBoostingRegressor	0.173	0.2607
XGBRegressor	0.1732	0.1341
RandomForestRegressor	0.1738	0.2257
ExtraTreesRegressor	0.1845	0.1024
DecisionTreeRegressor	0.2302	0.0144
AdaBoostRegressor	0.3154	0.2014
LinearRegression	0.3646	0.0043
KNeighborsRegressor	0.4916	0.0117
LinearSVR	1.0336	0.3175

Table 5.1: Performance of regression techniques on running time prediction.

5.2.2 Anytime performance

To evaluate our surrogate model's anytime performance, we will use a learning curve. We split the dataset k times in $n - \frac{n}{k}$ training samples and $\frac{n}{k}$ test samples. Then we randomly sample subsets of varying sizes of the training data to be used to train the estimator, and a score for each training subset size and test set will be computed. Afterwards, the scores will be averaged over all k runs for each training subset size.

We found that a gradient boosting model with a more restricted number of leaves and some regularization works better for small data. We found that more complicated models, where we allow for larger trees, need more data to work well. We show the results of a restricted and a more complex model in Figure 5.1. Using these models in Bayesian Optimization could therefore, especially in the beginning, lead to BO choosing evaluations that are more expensive to evaluate.



(a) Learning curve of LightGBM with 100 estimators, 128 leaves, learning rate of 0.1 and GBDT boosting.

(b) Learning curve of LightGBM with 100 estimators, 4 leaves, learning rate of 0.2 and GBDT boosting.

Figure 5.1: Learning curves of two LightGBM models. The y-axis shows negative MSE-loss (so higher is better) and the x-axis the number of training samples.

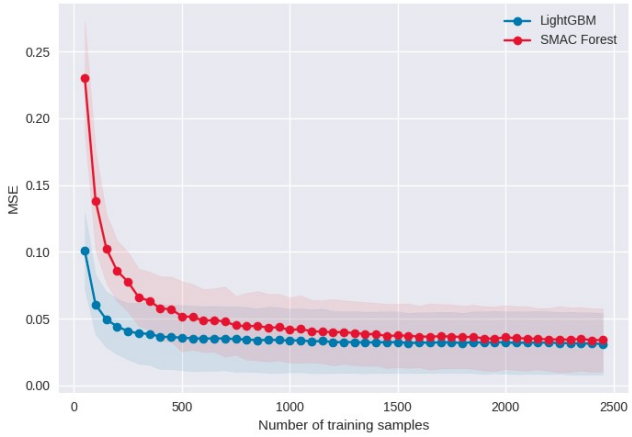


Figure 5.2: Runtime prediction performance of LightGBM and SMAC's Random Forest.

5.2.3 Results

Here, we will again use learning curves to compare our model with SMAC's model for predicting evaluation cost. We again split each dataset k times in $n - \frac{n}{k}$ training samples and $\frac{n}{k}$ test samples. And then we randomly sample subsets of varying sizes of the training data to be used to train the estimator, and the MSE loss for each training subset size and test set will be computed. We then take the average performance over 10 datasets, which are described in Table 4.1

Figure 5.2 shows that our LightGBM model with 100 estimators, 4 leaves, learning rate of 0.15, and $\alpha = 0.20$ outperforms SMAC's forest especially when we only have few samples to train on. Using this model, we can then produce better estimates for the cost of evaluations earlier in the Bayesian Optimization process. This leads to a faster start, and hence we find better configurations earlier.

Furthermore, EI/s also also helps in producing faster models, and so getting better estimates with less data also helps with producing faster models when the total time budget for Bayesian Optimization is low.

5.3 Quantify promising configurations

Finding a good estimator for performance prediction is a more difficult problem than finding an estimator for runtime prediction. And since this model not only needs to give a point estimate, but also a degree of how confident the model is about this estimate, i.e. the predictive uncertainty, we will setup a simulation and let the model choose the data itself.

5.3.1 Simulation

The simulation starts with 3 randomly chosen parameters, which the score estimator will be trained on initially. The score estimator then predicts the performance of the remaining hyperparameter settings, and we determine the setting with the highest expected improvement. This value will be evaluated, and the setting and its performance will be added to the set of observed settings. We then retrain the model on the observed settings, and predict the performance again of the remaining hyperparameter settings. This continues until we have simulated 250 iterations.

We used data of 40.000 runs on the model described in section 4.3, with different hyperparameter configurations across 10 different datasets as shown in Table 4.1.

5.3.2 Uncertainty for gradient boosting

Whereas Random Forests can calculate predictive uncertainty by using one of the modifications as described in section 3.2, this will not work for gradient boosting techniques.

In gradient boosting, each new tree helps to correct errors made by previously trained tree. So that usually makes the first tree contribute the most towards the final estimate, while the later trees are correcting the earlier trees' estimates.

Quantile regression

Instead, we can use quantile regression to quantify the uncertainty. Given a random variable Y (such as the predicted performance) and a value $p \in [0, 1]$, the associated quantile α, is the value such that $P(Y \leq p) = \alpha$. Figure 5.3 shows an example of quantile regression. Note that each quantile needs its own estimator.

It is possible to approximate the standard deviation of an estimate. For example, we can predict the 16th quantile (l_θ), 50th quantile (m_θ) and 84th quantile (u_θ) for a configuration θ and approximate our uncertainty for the point estimate m_θ as follows:

$$\sigma_\theta = \frac{u_\theta - l_\theta}{2} \tag{5.1}$$

We found however, that we often could not create a model for the 84th quantile, because all the target values above the quantile were the same (so the decision trees can not decide on splits). The calculation for the EI would then still work, but uses only the 16th quantile for approximating the uncertainty, which reinforced the problem of all the target values above the 84th being the same.

Upper bound estimation

Since we want to find the best hyper parameter setting, and therefore more interested in the upper bound of uncertainty than the lower bound, we drop the model for the lower bound. And because this upper bound model already gives an upper estimate, we do not need a point estimate anymore.

We will use these upper bound estimations to quantify directly how promising a configuration is. We can then replace the Expected Improvement by these upper bound estimations.

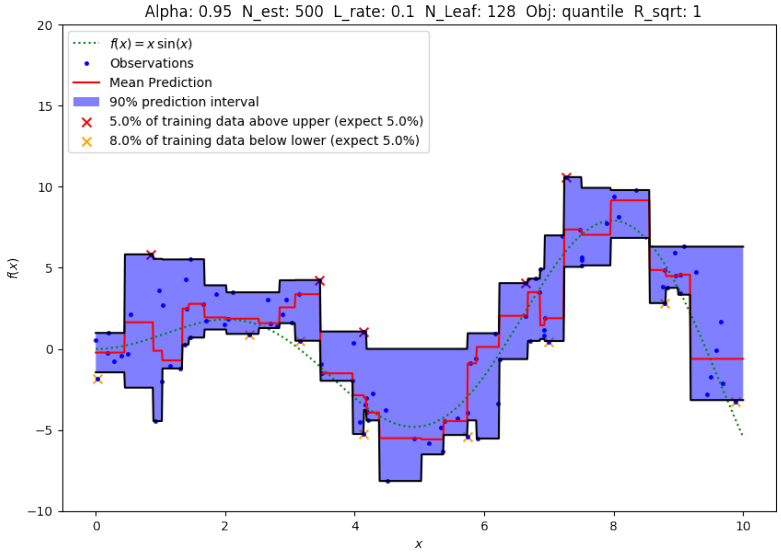


Figure 5.3: A regression interval created by quantile regression.

Implementation

We use LightGBM as estimator, setting the number of leaves to 8, and the number of estimators to 100. We set the estimator's objective to quantile regression and let the quantile equal $\alpha = 0.9$.

5.3.3 Results

Figure 5.4 shows our proposed method of Gradient Boosting with upper bound estimation using Quantile Regression (GBQR). We compare its performance against SMAC's surrogate model which is used for modeling the Expected Improvement.

For each iteration, we calculated the accumulated maximum (i.e. the best performance so far) as a fraction of the optimum. The optimum is defined as the maximum of our available data. The graph represents the mean over all tasks, while the area shows the standard deviation over these tasks.

GBQR found the optimum of all 10 datasets within 120 iterations, whereas SMAC's Random Forest found the optimum in only 2 datasets after 250 iterations.

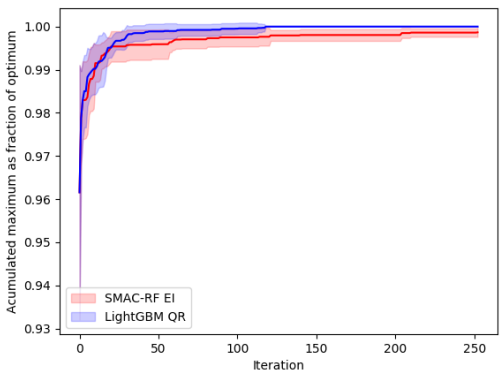
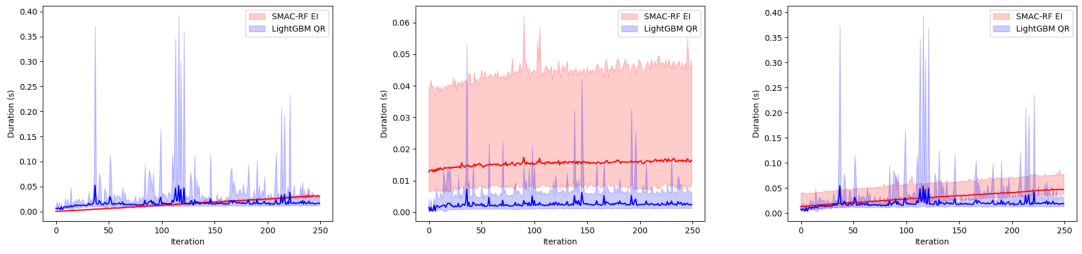


Figure 5.4: LightGBM with quantile regression compared with SMAC's Random Forest.

Our method is generally faster during prediction, but training time can sometimes spike. To explain these spikes, we have to note the following statements: (1) gradient boosting is sequential, but LightGBM parallelized node-building and (2) we have limited the number of leaves instead of the maximum depth. Because of (2), the trees can still grow deep and because of (1) in the worst case, where the tree only grows in depth, LightGBM can only use one thread to build a tree. Note that it is possible to solve this problem by setting a maximum depth for the trees instead of a maximum number of leaves.

Even so, the average training time seems to scale better if more data is added. Figure 5.5 shows the minimum, maximum and mean duration in seconds over the number of iterations. The total time represents the overhead of training the model and calculating the expected improvement or upper bound.



(a) Training time (over observed data) (b) Prediction time (over all data) (c) Total time

Figure 5.5: Comparison of training and prediction times of LightGBM and SMAC-RF.

5.4 Combined estimation

In this section we will combine our improvements of quantifying promising configurations and estimating evaluation cost. First we will show a modification for GBQR in subsection 5.4.1 so that we can combine it with the evaluation cost estimates. We explain the setup of the experiment in subsection 5.4.2, discuss the results in subsection 5.4.3 and finally discuss these results a bit more in subsection 5.4.4.

5.4.1 Evaluation cost for GBQR

GBQR calculates an upper confidence bound on the performance over all configurations instead of an expected improvement on the current optimum (incumbent). The idea of calculating how much progress we can expect to make at a certain cost, such as EI/s does, is therefore not possible in GBQR without some modifications.

Note furthermore that the optimum is larger than the 90% upper confidence bound estimates, as the optimum itself is in the 100th percentile. We can therefore not simply subtract the current optimum (incumbent) from the upper bound estimates.

After some experimenting, we found that normalizing the upper confidence estimates by subtracting the median of these estimates (and then dividing by the expected log-time) worked reasonably well. More formally, we quantify the value Q_θ of selecting a configuration θ as follows:

$$Q_\theta = \frac{\upsilon_\theta - \tilde{\upsilon}}{c_\theta} \tag{5.2}$$

Here, c_θ is the evaluation cost defined as the predicted duration of evaluating θ in log-transformed seconds: $log(s)$. And further, υ_θ is the upper confidence bound estimate for θ and $\tilde{\upsilon}$ is the median of the upper confidence bound estimates on all $\theta \in \Theta$, where Θ is the set of all configurations that we are considering (in the current iteration).

Parameter	Range
n_estimators	[100]
learning_rate	[0.001, 0.0055, 0.01, 0.055, 0.1]
num_leaves	[4, 8, 16, 32, 64, 128]
reg_alpha	[0, 0.1, 0.2]
reg_lambda	[0, 0.1, 0.2]
min_child_samples	[1, 20, 100]
max_depth	[4, 6, 8, 10, 12]

Table 5.2: LightGBM parameter grid.

5.4.2 Experiment setup

Now that we have shown the performance of our new methods individually on a set of 10 tasks, we will increase the amount of tasks and re-evaluate these performances using a LightGBM target model. We will first execute this new target model on these tasks using different configurations on the new tasks, and store the configurations along with the resulting performance and running time. In this manner, we will not have to evaluate these again while testing our different methods.

Target model

We will use LightGBM as our target model. Table 5.2 shows the parameter grid that we have used for our executions on the new set of tasks. For each task, we have run all possible combinations of this grid, i.e. that is 4050 runs per task.

Tasks

Table 5.3 gives an overview of the tasks we will use to evaluate or methods. These tasks are selected from OpenML CC18[1], a benchmark suite curated by OpenML. We have omitted tasks for which $\#\text{samples} * \#\text{features} > 9,000,000$ in order to reduce computing time, and by accident we only selected binary classification tasks, although this should not matter for our experiments. Running all configurations for each task gives us a total of 125,550 runs over 31 datasets.

Interleaving with randomized search

According to the authors of SMAC, the Random Forests' ability to interpolate is insufficient. To address this problem, they proposed to evaluate one random configuration every other iteration. This would help avoid the optimization process to end up in local optimum. Since we have not addressed this before, we thought it was a good idea to include this proposal here.

5.4.3 Results

We will first compare all the methods that do not perform cost estimation. After that, we will compare all the methods with cost estimation, except the GBQR-based method. And finally, we will compare the best methods against each other.

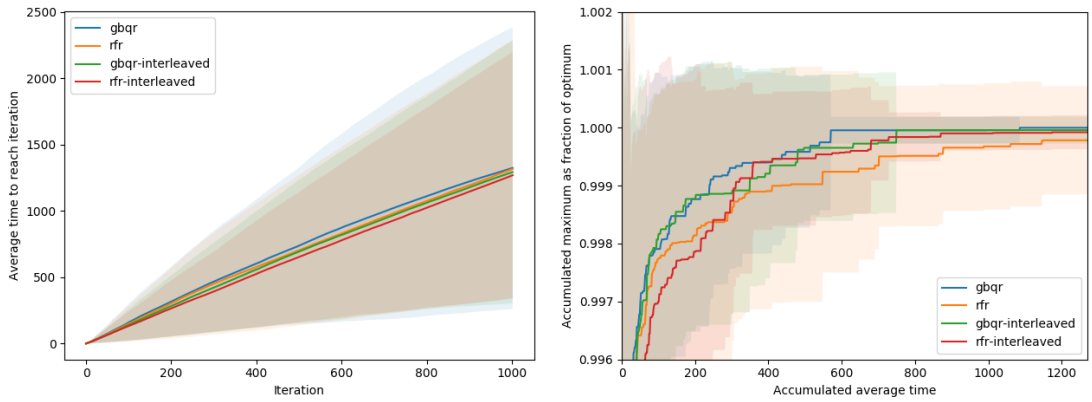
Methods without cost estimation

Figure 5.6 shows that interleaving the optimization process with random evaluations works quite well for SMAC's Random Forest. However, for our own method, we do not see any benefit: the normal GBQR still outperforms the other methods.

[1] https://www.openml.org/s/99

task	dataset name	# samples	# features	# classes
3	kr-vs-kp	3196	37	2
15	breast-w	699	10	2
29	credit-approval	690	16	2
31	credit-g	1000	21	2
37	diabetes	768	9	2
43	spambase	4601	58	2
49	tic-tac-toe	958	10	2
219	electricity	45312	9	2
3021	sick	3772	30	2
3902	pc4	1458	38	2
3903	pc3	1563	38	2
3904	jm1	10885	22	2
3913	kc2	522	22	2
3917	kc1	2109	22	2
3918	pc1	1109	22	2
7592	adult	48842	15	2
9946	wdbc	569	31	2
9952	phoneme	5404	6	2
9957	qsar-biodeg	1055	42	2
9971	ilpd	583	11	2
9978	ozone-level-8hr	2534	73	2
10093	banknote-authentication	1372	5	2
10101	blood-transfusion-service-center	784	5	2
14952	PhishingWebsites	11055	31	2
14954	cylinder-bands	540	40	2
14965	bank-marketing	45211	17	2
125920	dresses-sales	500	13	2
146818	Australian	690	15	2
146819	climate-model-simulation-crashes	540	21	2
146820	wilt	4839	6	2
167141	churn	5000	21	2

Table 5.3: Overview of tasks used for experiments with LightGBM configurations

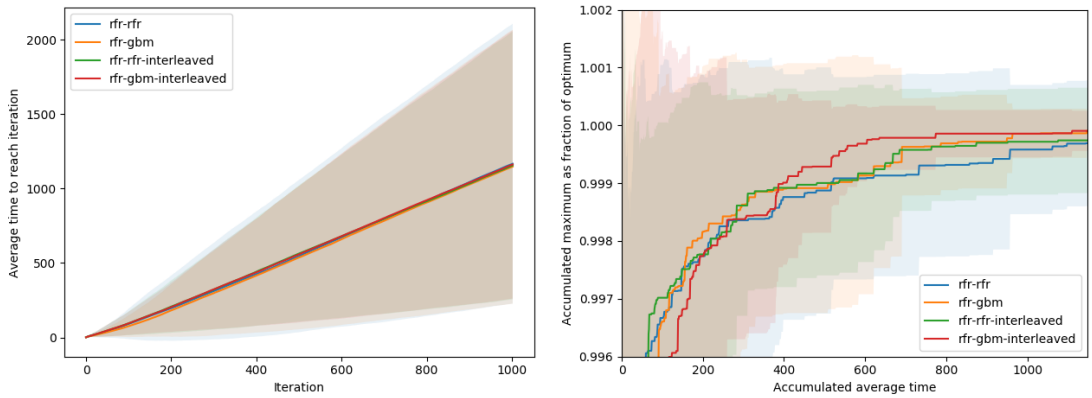


(a) Speed (b) Performance

Figure 5.6: Speed and performance of SMAC's forest and GBQR interleaved with randomized search.

Methods with cost estimation

Figure 5.7 shows that interleaving also works well for methods that include run time estimation. SMAC's Random Forest (RFR) combined with our gradient boosted time-estimator seems to work best, and interleaving seems to work well too here. The total evaluation time for each of these methods is however very similar. Comparing to Figure 5.6 where the interleaved methods took less evaluation time, we can derive that in this case, GBQR and RFR alone tend to find configurations that take more time than the average configuration. In combination with a time estimator, the chosen configurations take similar time to evaluate compared with the average configuration.

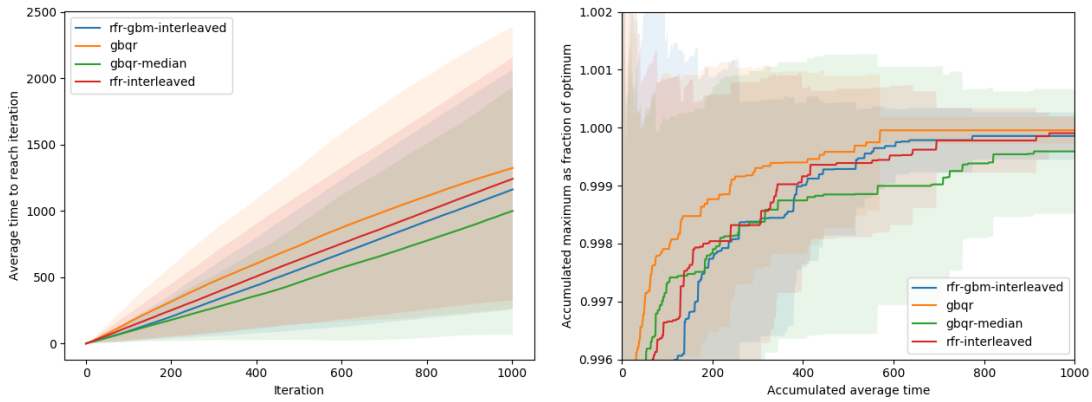


(a) Speed (b) Performance

Figure 5.7: Speed and performance of SMAC's forest including time-estimation and interleaving.

Comparison of best performing methods

Figure 5.8 shows a comparison of the best RFR methods against our GBQR methods. We see that GBQR performs the best, while GBQR-median does not perform as well as the other methods. Since RFR-interleaved and RFR-GBM-interleaved also perform similar, we think that using a combined performance and run time estimation does not work very well.



(a) Speed (b) Performance

Figure 5.8: Speed and performance comparison of the best methods.

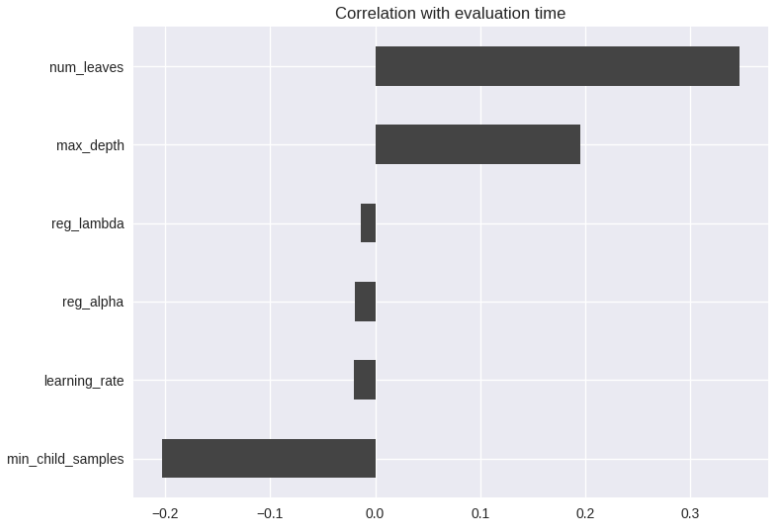


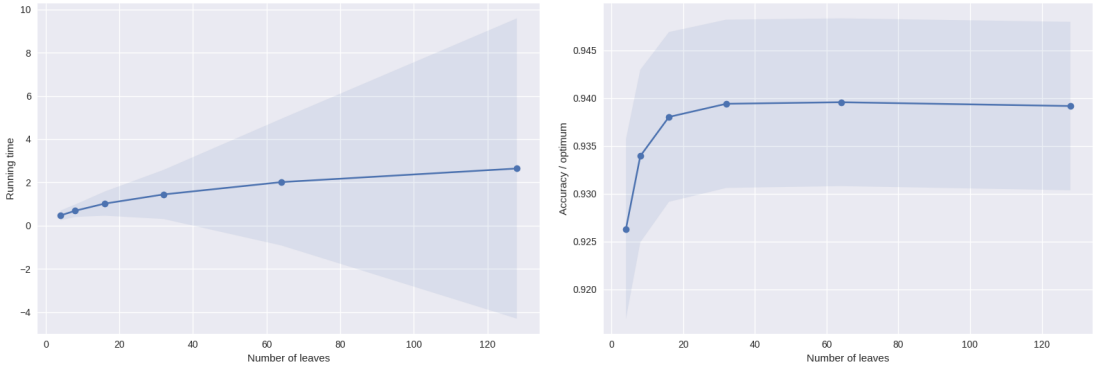
Figure 5.9: LightGBM parameter correlation with running time.

5.4.4 Discussion

We have shown that GBQR works well for optimizing a LightGBM model, and we have measured the GBQR performance on a larger set of tasks. We did however notice that the methods that combined performance and run time estimation did not work as well as expected. In section 5.4.3 we observed that GBQR and RFR tend to find configurations that take more time than the average configuration, while the combined methods choose configurations with similar evaluation time as the average configuration.

From this we can conclude that expected improvement per second (and similar methods) does not always converge to the optimum faster (although it might still help in finding a faster solution). In our case, the number of leaves in a model strongly correlates with the running time of the model (see Figure 5.9). However, adding more leaves to our model increases the performance with diminishing returns (see Figure 5.10).

The expected improvement per second therefore puts too much preference on shallow trees, while more complex trees might result in better performance.



(a) Speed (b) Performance

Figure 5.10: Speed and performance by number of leaves.

Chapter 6

Improving the warm-starting of Bayesian Optimization

When applying Bayesian Optimization to a dataset, the optimization process starts with zero evaluations, and thus zero knowledge. However, with meta-learning we can transfer knowledge gained from one dataset to the other.

Suppose that we are given a collection of datasets on which hyperparameters are already tuned by either humans with domain expertise or extensive trials of cross-validation. When a model is applied to a new dataset, it is desirable to let Bayesian optimization start from configurations that were successful on similar datasets. This practice is called warm-starting.

For example, AutoSklearn uses warm-starting as follows. Given a new dataset D, AutoSklearn computes its meta-features, ranks all seen datasets by their L_1 distance to D in meta-feature space and selects the best machine learning instantation for each of the $k = 25$ nearest datasets for evaluation before starting Bayesian optimization with their results.

Another approach is to train a meta-model on the large collection of datasets and hyperparameters, and then use use this model to help decide which configuration to evaluate next. In the beginning this meta-model will have a large vote in which configuration to evaluate next, and over time the surrogate model will gain a larger vote on this matter.

In this chapter, we will look at machine learning algorithms that are able to create a good meta-model. We will call these algorithms meta-learners.

6.1 Meta data

Meta data is the data that we feed into the meta-learner. This data consists of information about each run (hyperparameters) and the dataset that the algorithm is run on (metafeatures). For each machine learning pipeline (i.e. a "flow"), we create a single meta-learner. The flows used in our experiments are described in subsection 6.1.1, their hyperparameters in subsection 6.1.2 and the meta-features for the datasets in subsection 6.1.3.

6.1.1 Flows

OpenML has many runs submitted by users and bots that we can compare, study and reproduce. We are going to take a closer look at five different flows for classification. Table 6.1 shows a description of each flow. These flows are all compatible with the Scikit-learn package and were chosen based on the number of runs. Flows that are based on the same pipeline, and Randomized Search flows, were not taken in consideration.

Flow	# Runs	# Datasets	Description
6840	14,150	59	Decision stumps with simple imputing (mean, median or most frequent).
6969	132,613	84	A random forest with 100 trees, imputing for nominal values (most frequent) and numerical values (mean, median or most frequent), one-hot-encoding, and a feature-selector that remove all zero-variance features.
8315	50,869	51	A random forest with 500 trees, imputing for nominal values (most frequent) and numerical values (mean, median or most frequent), one-hot-encoding, and a feature-selector that remove all zero-variance features.
6970	71,497	84	Adaboost with 50 to 500 trees, imputing for nominal values (most frequent) and numerical values (mean, median or most frequent), one-hot-encoding, and a feature-selector that removes all zero-variance features.
8317	59,164	52	SVM with the RBF kernel, imputing for nominal values (most frequent) and numerical values (mean, median or most frequent), one-hot-encoding, feature standardizing, and a feature-selector that removes all zero-variance features.

Table 6.1: Description of the five flows used in this experiment.

6.1.2 Hyperparameters

OpenML stores all the hyperparameters that are used during a run. We retrieve these by querying the ElasticSearch service running on the OpenML servers. After converting this data to a usable format, we need to convert the hyperparameters to numerical data so that it can be used with regression techniques. This process is described in section 3.1. Table 6.2 shows the hyperparameters and their ranges for each flow.

6.1.3 Metafeatures

OpenML provides many meta-features for each dataset. We use a subset of these meta-features that excludes meta-features that are only applicable to numerical features (for example skewness). The reasoning is that not all features are numeric, so it will be hard to predict what the impact is, even if we know the number of numeric features. And not every dataset might include numeric features. Furthermore, we exclude meta-features such as "number of binary features" and "number of instances with missing values" because these are correlated with the number of features or instances. We will however make use of their percentage-equivalents, such as "percentage of binary features" or "percentage of instances with missing values". The statistical metafeatures used in the experiments can be seen in Table 6.3.

Next to the statistical metafeatures, OpenML provides many land-markers based on the Weka machine learning package. Table B.1 provides a full list of land-markers used in the experiments. Before we use the metafeatures, we standardize all the values.

Flow	Parameters	Range
8317	C	(0.03125, 32763)
	coef0	(0, 1)
	gamma	(0.00003, 8)
	shrinking	[False, True]
	tol	(0.00001, 0.1)
	nominal imputing	[most_frequent]
	numerical imputing	[mean]
6970	algorithm	[SAMME, SAMME.R]
	learning_rate	(0.01, 2.00)
	n_estimators	[50, ..., 500]
	max_depth	[1, ..., 10]
	max_features	(0.02372, 0.044721)
	nominal imputing	[most_frequent]
	numerical imputing	[mean, median, most_frequent]
6969	bootstrap	[False, True]
	criterion	[gini, entropy]
	max_features	(0.04, 0.90)
	min_samples_leaf	[1, ..., 20]
	min_samples_split	[2, ..., 20]
	nominal imputing	[most_frequent]
	numerical imputing	[mean, median, most_frequent]
6840	criterion	[entropy, gini]
	min_samples_leaf	[1, ..., 20]
	min_samples_split	[2, ..., 20]
	imputing	[mean, median, most_frequent]
8315	bootstrap	[False, True]
	criterion	[gini, entropy]
	max_features	(0, 1)
	min_samples_leaf	[1, ..., 20]
	min_samples_split	[2, ..., 20]
	nominal imputing	[most_frequent]
	numerical imputing	[mean]

Table 6.2: Parameters of the five flows used in this experiment.

Metafeature	Description
NumberOfInstances	Number of instances (rows) of the dataset.
NumberOfFeatures	Number of attributes (columns) of the dataset.
NumberOfClasses	Number of distinct values of the target attribute (if it is nominal).
Dimensionality	Number of attributes divided by the number of instances.
ClassEntropy	Entropy of the target attribute values.
PercentageOfBinaryFeatures	Percentage of binary attributes.
PercentageOfInstancesWithMissingValues	Percentage of instances having missing values.
PercentageOfMissingValues	Percentage of missing values.
PercentageOfNumericFeatures	Percentage of numeric attributes.
PercentageOfSymbolicFeatures	Percentage of nominal attributes.
MajorityClassPercentage	Percentage of instances belonging to the most frequent class.

Table 6.3: List of statistical metafeatures used for meta-learning.

6.2 Experiments

To get an idea of how well we can estimate the score for a given algorithm configuration, we will perform a series of experiments.

6.2.1 Methods

Our goal is to estimate configuration performance for a set of four machine learning flows. We will use both hyperparameters and metafeatures to estimate this performance on a set of four different metrics.

Types of input

Each run in OpenML has some performance metrics (for example, accuracy). We map the data associated with each run to one of these metrics. For example, let Θ be the hyperparameter configuration of each run, and let Y be the target value (for example, accuracy) of each run. Then, $\Theta \rightarrow Y$ is a simple mapping from hyperparameters to accuracy.

Now, let M_r be the set of metafeatures of the dataset on which run $r \in R$ is executed. We then define $\Xi = \{M_r | r \in R\}$. And we also define operator $\oplus$ as a column-wise concatenation.

Then, for our experiments we use the following four (sub)sets of $\Theta \oplus \Xi$:

- $\emptyset$ or **none:** A baseline where no input features are given. We simply predict the same value for all test data. This value equals the mean target value of the training set.
- Ξ or **metafeatures:** Only the meta-features of each dataset are given. These are the same for all runs on one dataset. Therefore, our predictions for the runs executed on the same dataset are also equal.
- Θ or **hyperparameters:** Only the hyperparameters of each run are given.
- $\Theta \oplus \Xi$ or **both:** Both hyperparameters and metafeatures are given for each run. These are concatenated together.

Meta-learner

We will use a LightGBM model to predict the performance of a hyperparameter configuration on a dataset. This model is manually tuned to work well with metafeatures as input. We use 500 trees, with a learning rate of 0.05 and a maximum of 4 leaves per tree.

Metrics

We will predict configuration performance on four different metrics: accuracy, Cohen's kappa, f1-metric and area under ROC curve. As Kappa's domain is $(-1, 1)$ rather than $(0, 1)$, the domain is transformed first to a $(0, 1)$-domain to make Kappa comparable with the other metrics.

Leave one dataset out

To validate our estimator, we use a Leave One Group Out cross-validator. Each iteration, we use one dataset to test on, and the remaining datasets will be used as training data. The number of iterations here equals the total number of datasets: we validate the estimator once per dataset.

Standardized metrics

We run our experiments again using metrics that are standardized per dataset. We group all runs together by the dataset they were performed on, and then standardize their performance.

6.2.2 Kendall's tau

Rather than predicting the performance of each run precisely, we would like to be able to distinguish low-performing hyperparameter settings from high-performing settings. This means that, for an unseen dataset, our predictions can be off on average, as long as we predict the high-performing settings to have a higher performance than the low-performing settings.

To measure this performance, we can calculate the Kendall rank correlation coefficient, commonly referred to as Kendall's tau. Kendall's tau measures the similarity of the orderings of the data when ranked by each of the quantities. We will use the "tau-b" version of Kendalls tau, which can account for ties and which reduces to the "tau-a" version in absence of ties.

The definition of Kendall's tau that is used is

$$\tau(x, y) = \frac{P - Q}{\sqrt{(P + Q + T) * (P + Q + U)}} \tag{6.1}$$

where P is the number of concordant pairs, Q the number of discordant pairs, T the number of ties only in x, and U the number of ties only in y. If a tie occurs for the same pair in both x and y, it is not added to either T or U.

We can use Kendall's tau to measure the rank correlation between the meta-model's predicted algorithm performances for different settings on a dataset and their true performances. Using this measure, we are putting more focus on the meta-learner's ability to distinguish good settings from bad settings.

6.3 Experiment results

We show both error and correlation for each flow and each metric in Figure 6.1. Furthermore, we run these experiments again on standardized metrics, which results are compared with the non-standardized metrics in Figure 6.2

6.3.1 Loss and correlation

We can see that hyperparameters alone can not predict the precise performance very well. Metafeatures on the other hand predict this precise performance much better.

It could be that meta-features are more important for estimating the mean performance, while the combination with parameters is needed for distinguishing high-performing settings from low-performing settings.

We can confirm this by calculating the standard deviation of each dataset's mean performance per dataset and comparing it with the mean of the performance's standard deviation per dataset. For flow 6969, we see that the spread over all datasets is 7-12 times bigger than the spread per dataset.

This shows that the variance within one dataset is significantly lower than the variance between datasets. So the performance of each setting within one dataset must be somewhat close together. And that means that the dataset type (i.e. the dataset's properties which are expressed as meta-features) itself has the most impact on the mean performance.

Interestingly enough, for flow 8317, the spread over all datasets is only 1.1-1.3 times bigger than the spread per dataset. This indicates that in this case, the hyperparameters contribute a lot more towards the performance of a run, compared to other flows (note that increasing the parameter space of other models would achieve a similar effect). This also reflects in the graph, which shows more difference between the "none" bar and the "hyperparameters" bar, while for flows 8315 and 6969, we see very little difference between these two bars.

We can furthermore see that combining metafeatures and hyperparameters is often similar to the combined difference between (i) predicting with hyperparameters and the baseline "none", and (ii) predicting with metafeatures and the baseline "none". However, in the correlation graphs, we

can see that this combination helps ranking the configurations much better than hyperparameters alone. This shows that the estimator is able to understand the relation between hyperparameters and metafeatures.

6.3.2 Standardized metrics per dataset

Using standardized metrics, we force the estimator to forget about differences per dataset and instead focus on high-performing and low-performing configurations, and the interaction between configurations (hyperparameters) and the properties of the dataset (metafeatures).

This is very similar to ranking approximated by a regression problem, in a sense that all configurations are given a numerical score based on how well they perform within one group. We therefore also experimented with the LGBMRanker method, which learns a pairwise ranking for each dataset. However, this method showed lower performance and is a lot slower, because every configuration needs to be checked against the other in the group.

In Figure 6.2 we can see that standardizing improves correlation using both hyperparameters and metafeatures as estimator input. Using a more complex model with a maximum of 16 leaves can sometimes improve the correlation even more when both hyperparameters and metafeatures are taken as input. This seems to work at the cost of the performance when we have only hyperparameters as input.

6.3.3 Feature importances

We have plotted the feature importances of each flow in Figure 6.3. These feature importances are for predicting standardized accuracy.

We can see that usually, the hyperparameters of the model are also the most important features. For the metafeatures, "Autocorrelation" and "ClassEntropy" seem to be important for all flows. These measure, respectively, the average class difference between consecutive instances and the entropy of the target attribute values. That is, they give some measure of difficulty of the task.

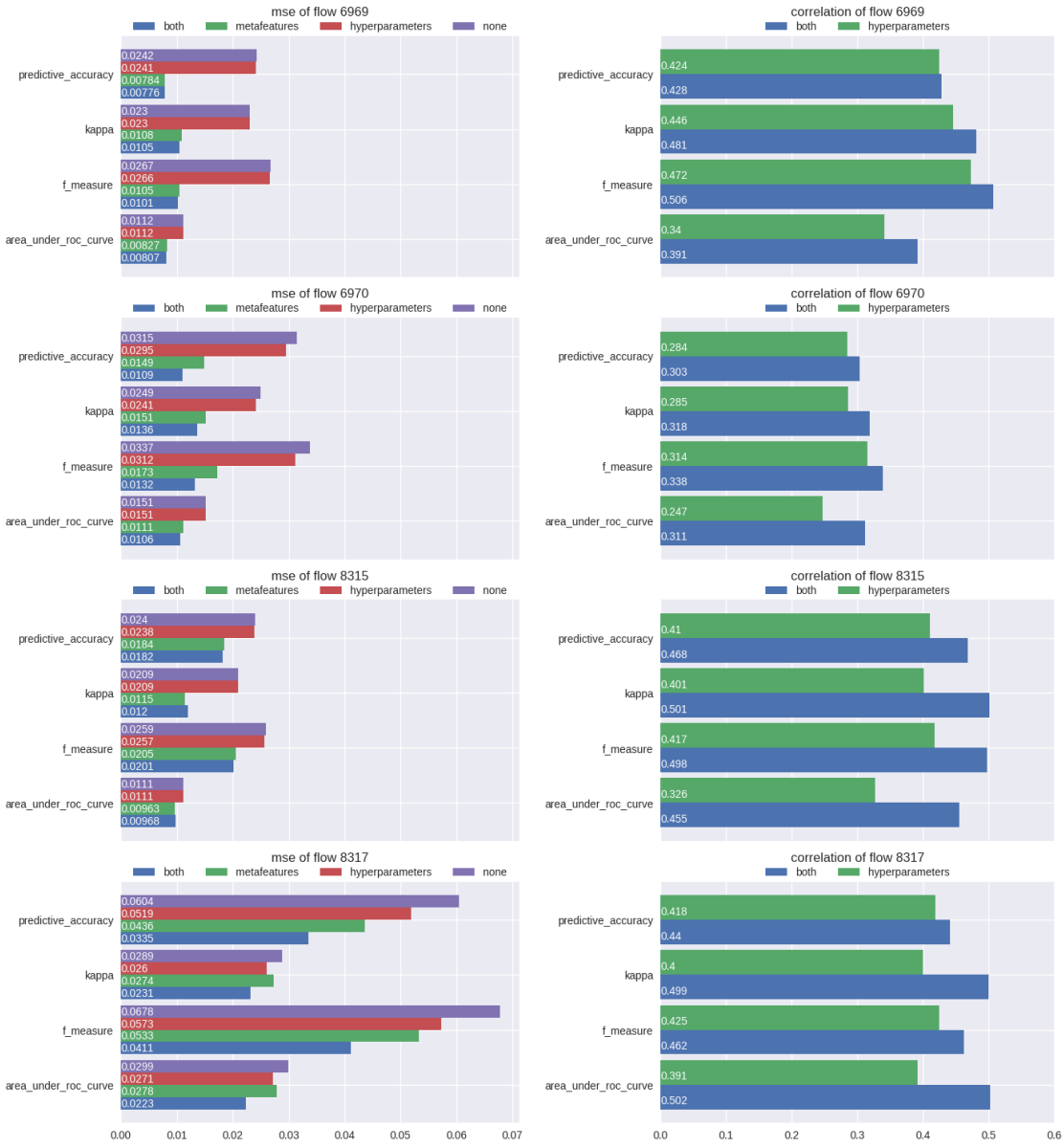
For the tree-based models, we see that "NumberOfFeatures" is important. This metafeature is important, because it relates to the hyperparameter "max_features", which determines the ratio of features that is selected. "NumberOfFeatures" gives the exact number of features.

6.3.4 Comparison with KNN

Since AutoSklearn uses KNN to find the 25 most similar datasets, we think it is interesting to look at KNN as a baseline. In this way we can also find out how much we can learn from looking only at similar datasets.

Missing values in the meta-features are imputed using the median and these metafeatures are then standardized. The parameters are also standardized. Then, for a given datasest, we find the k most similar datasets using unsupervised KNN. We found that $k = 1$ works the best here. We then use supervised KNN to find the 5 closest runs, weighted by distance, for each configuration we want to predict over. Alternatively, we can use LightGBM to train on this dataset, and predict on the new dataset. We will call this method "KNN+LightGBM".

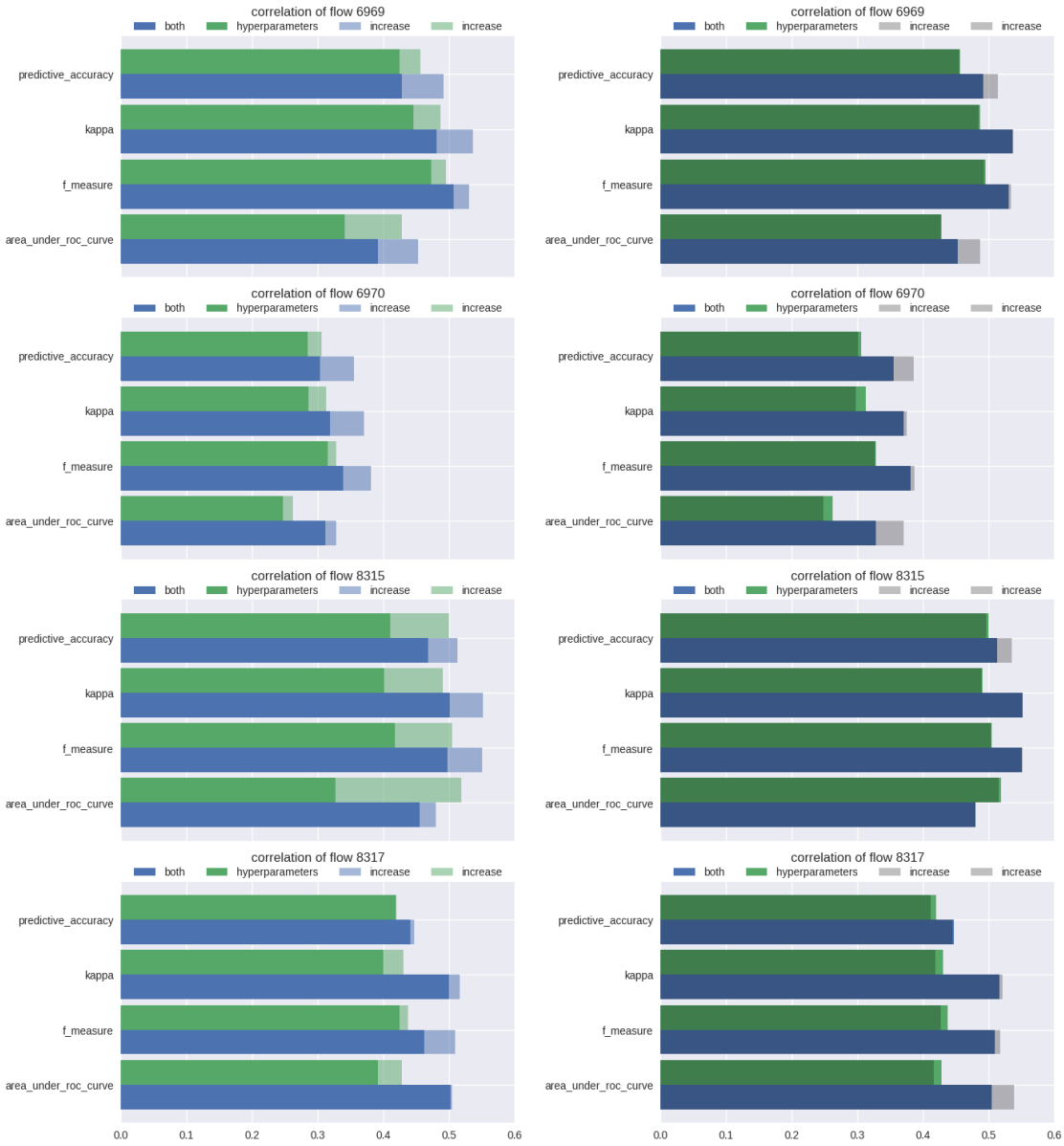
Figure 6.4 shows that our meta-learner "LightGBM" is significantly better than the KNN meta-learning method.



(a) Mean squared error (lower is better)

(b) Correlation (higher is better)

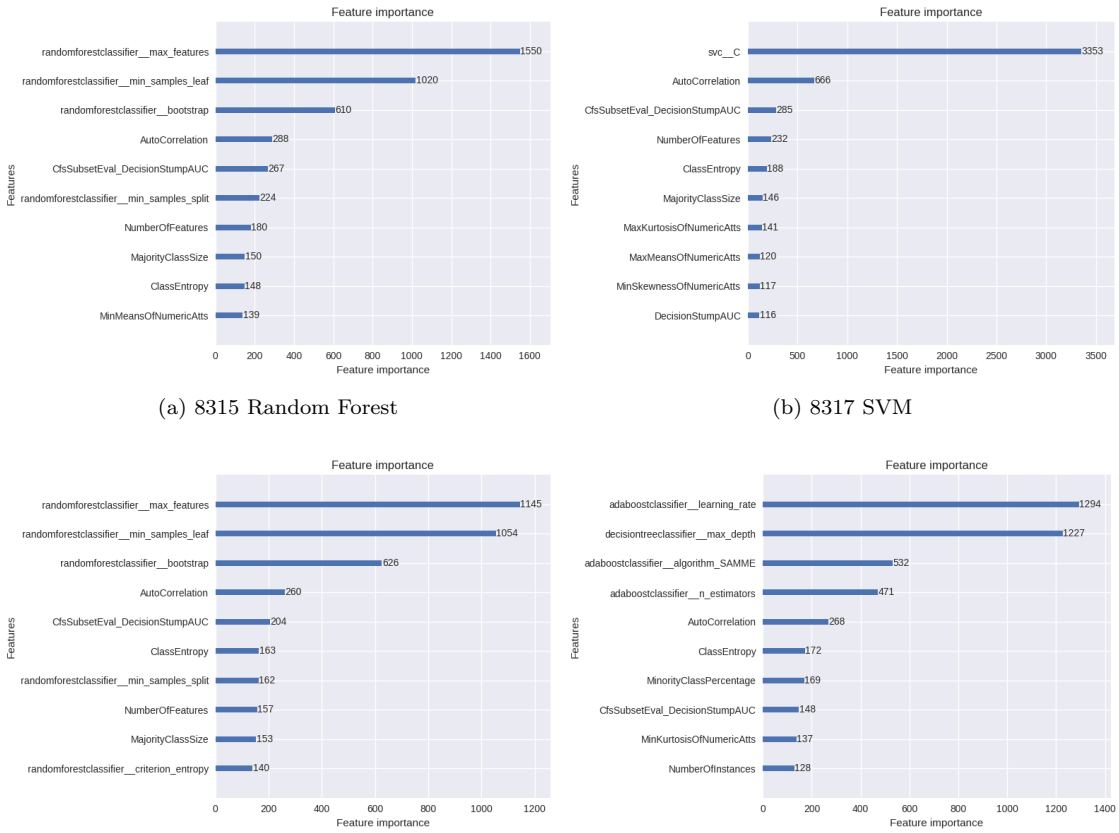
Figure 6.1: Estimation error and rank-correlation.



(a) After standard-scaling

(b) Using deeper trees

Figure 6.2: Increase in correlation performance (higher is better) after standard-scaling the metric, and after standard-scaling and using a more complex model.



(a) 8315 Random Forest

(b) 8317 SVM

(c) 6969 Random Forest

(d) 6970 Adaboost

Figure 6.3: The feature importances of the 10 most important features for estimating predictive accuracy.

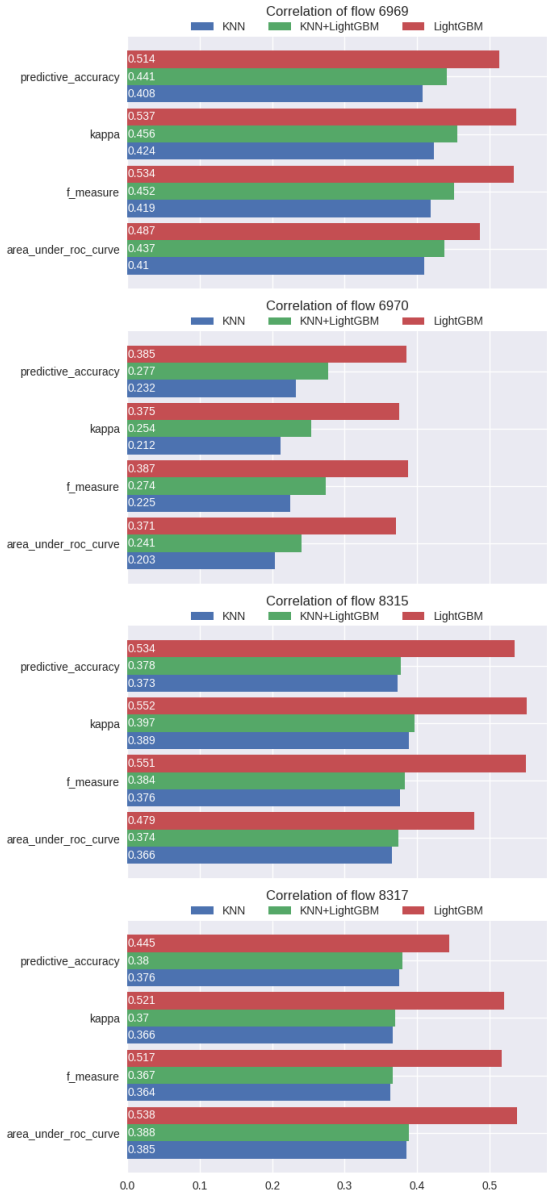


Figure 6.4: A comparison of KNN with LightGBM

6.4 Using the meta-learner for warm-starting

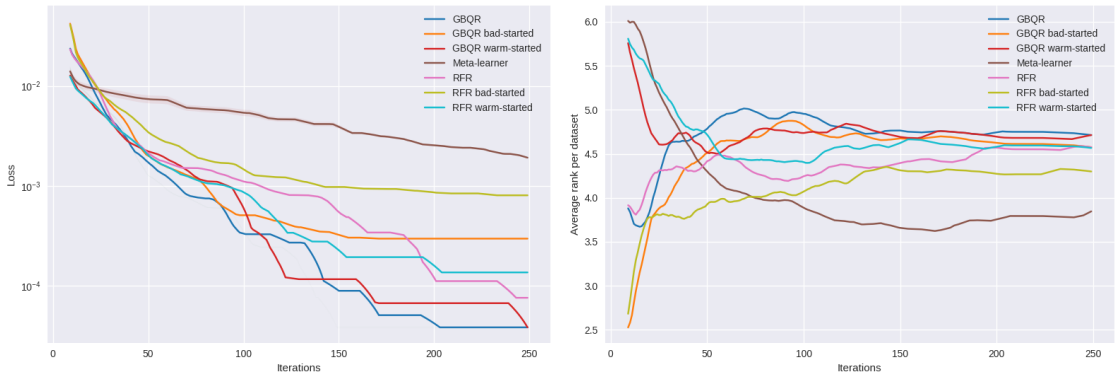
We use warm-starting using the three *best* predicted samples according to the meta-learner. For demonstration purposes, we will also show warm-starting using the three *worst* predicted samples, which we will dub "bad-starting". Next to that, we will also include a run with pure meta-learning, i.e. we follow the ranking order as predicted by the meta-learner, from best configuration to worst configuration.

We will use SMAC's forest with interleaving (here called RFR, the abbreviation that SMAC uses internally) as well as our GBQR method to compare our warm-starting methodds.

The experiments are executed using 56 datasets from flow 6969 (Random Forest). We have excluded the datasets for which we did not have landmarker metafeatures. We have chosen to use warm-starting with three samples as starting point, because the surrogate model is very quick to find better results than the meta-learner.

Figure 6.6 shows that bad-starting has a significant effect on the performance of the surrogate models. Warm-starting seems to help the RFR estimator stay ahead of the randomly initialized RFR in the beginning, while for our GBQR method we see not much difference.

Comparing the bad-started to the warm-started estimators, we can nevertheless see the need for a good initial set of configurations.



(a) Loss (lower is better)

(b) Ranking (higher is better)

Figure 6.5: Ranking and loss for warm-starting.

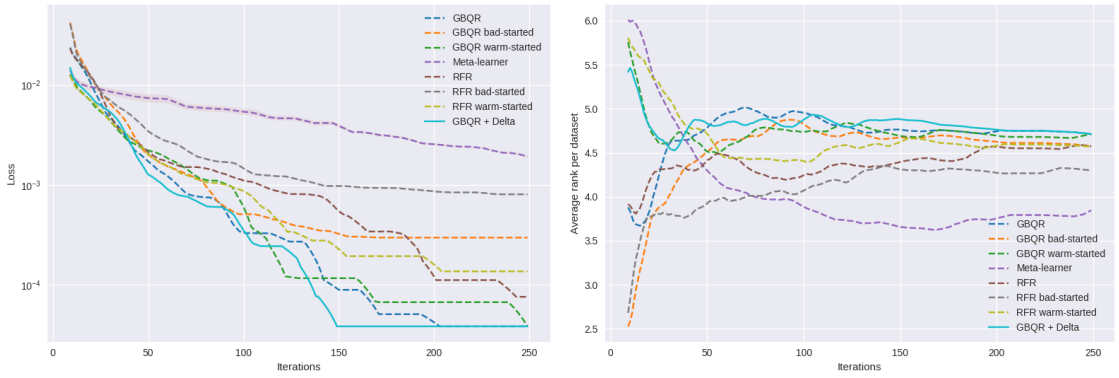
It is interesting to see that random initialization (RFR and GBQR) performs very similar to warm-starting using the three highest predicted configurations, and at the same time, bad-starting gives much worse results. An explanation for what is happening is that randomized initialization gives the surrogate model a wider spread in performance and more difference in hyperparameters for each sample. This then gives the surrogate model more information. Warm-starting on the other hand, is likely to start with hyperparameters that are very close. In particular, it is likely that only the least important hyperparameter varies in the top three highest predicted configurations.

To solve this problem, we thus need to vary our input more. However, note that a simple approach, such as using a triple with the best, median and worst predicted configuration for initalization would not work well. During testing we found that in nearly all cases, the meta-model's predictions were more correlated with the true performances than the surrogate model's predictions. This indicates that the surrogate model mostly only learns about a small part of the hyperparameter space, i.e. the set of highest performing configurations. Therefore, we need to produce a triple that is predicted to be perform well, so that our estimator has information about high-performing configurations, but we need some more spread in our hyperparameters and performance.

To increase the spread, we introduce a simple metric that takes the difference of performance with the highest estimate into account, as well as the estimated performance, as follows:

$$\texttt{score}(p_i, \Delta) = (1 - p_i)^{\Delta} + p_i \tag{6.2}$$

where $p_i \in P$ is a predicted value as estimated by the meta-learner, where P is scaled to a $(0, 1)$-range The parameter $0 \leq \Delta \leq 1$ regulates the importance of this difference. We used $\Delta = 0$, $\Delta = 0.1$ and $\Delta = 0.2$ and selected the samples with the highest score to include in our warm-starting triple. We call this method the delta-approach, which we will use together with our GBQR method. The results are shown in Figure 6.6.



(a) Loss (lower is better)

(b) Ranking (higher is better)

Figure 6.6: Ranking and loss for warm-starting using the delta-approach.

We see that our delta-approach outperforms the other methods, which shows the need for not only providing initial points that are (expected to be) high-performing, but also have some spread in the hyperparameters so the Bayesian Optimization surrogate model "knows" which hyperparameters make the most impact on performance. We introduced this spread via our delta-method, but there are other ways to reach this goal that are still left explored.

6.5 A comparison with Auto-Sklearn

We can now say that it is probably beneficial to not only store the best algorithm instantation for each dataset, such as AutoSklearn does, but to keep the results of many runs. Furthermore, we know that from looking only at similar datasets (which AutoSklearn also does), we are able to learn less than when we are learning from a large set of datasets and bundle hyperparameters and metafeatures together so that a meta-learner can learn the relations between these two. Our meta-learning methods are therefore likely to outperform AutoSklearn in performance, although we can not compare this directly with the data we have: AutoSklearn takes the 25 best performing *instantiations* (i.e. a pre-configured algorithm, or in other words: a CASH-configuration where only the parameters for the current algorithm are active) to warm-start multiple hyperparameter optimization processes, while we take the 3 best *configurations* to warm-start one hyperparameter optimization process.

Methods

To give a better comparison of how Auto-Sklearn's method compares against our warm-starting method, we will introduce two new methods:

- **KNN-3:** Select 3 most similar datasets with KNN based on metafeatures and then use the top-performing configuration of each dataset to warm-start.

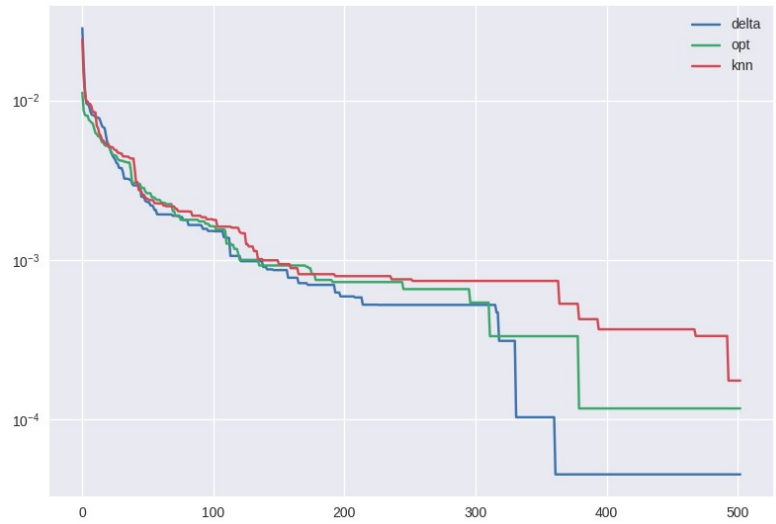


Figure 6.7: A comparison of Delta, OPT-3 and KNN-3. Loss is represented on the y-axis, while the number of iterations are represented on the x-axis.

- **OPT-3:** Use the top-performing configurations of the actual most similar 3 datasets. The definition of "actual most similar" is based on the correlation, between each pair of datasets, of the ranking of 4050 configurations executed across all datasets.

AutoSklearn selects a set of 25 instantiations, which can be interpreted as a set of configurations for a set of different algorithms, i.e. not all configurations are configurations for one and the same machine learning algorithm. So, we do not know which machine learning algorithms are going to be initialized and how many warm-starting configurations each algorithm will get.

Using our newly introduced methods, KNN-3 and OPT-3, we fix this number of warm-starting configurations to 3. We earlier reasoned that 3 is a good number, because the surrogate model is very quick to find better results than the meta-learner.

We can now see that KNN-3 is very similar to AutoSklearn. Next to that, OPT-3 provides as a theoretical optimum for KNN-3's ability to judge how similar a pair of datasets are.

Results

Figure 6.7 shows the results after using KNN-3, OPT-3 and our Delta method for warm-starting Bayesian Optimization with a LightGBM target model on 31 datasets. We can see that our Delta-method outperforms both KNN-3 and OPT-3.

Chapter 7

Exploring algorithm selection as a multi-armed bandit problem

In this chapter we will explore algorithm selection as a multi-armed bandit problem. We have a set of machine-learning algorithms that we want to optimize with limited resources. Our goal is to determine how many resources, in this case iterations, we allocate to the hyper-parameter optimization of which algorithm. We will treat this selection process as a multi-armed bandit problem.

We deviate here from solutions as AutoSklearn, which reduce the algorithm selection and hyperparameter optimization (CASH) problem to a hyperparameter optimization problem by concatenating the hyperparameters of multiple machine learning algorithms together, and using a boolean to indicate which algorithms are in use in the machine learning pipeline.

7.1 Problem setup

Figure 7.1 shows the setup of this problem. We have several competing BO-processes, which alternately decide on a promising configuration and then evaluate this configuration. Each BO-process is tied to a different machine learning algorithm. We then run a competition process with multiple rounds, where we decide per round whose turn it is, i.e. which BO-process can execute one iteration. We want to allocate these rounds strategically, such that we allocate more rounds to BO-processes that are likely to eventually produce the highest performing machine learning instantiation. We do this by using a bandit algorithm that decides on which lever to pull, based on the rewards it received for each BO-process. Here, the rewards are defined as the performance we observed from the evaluated configurations.

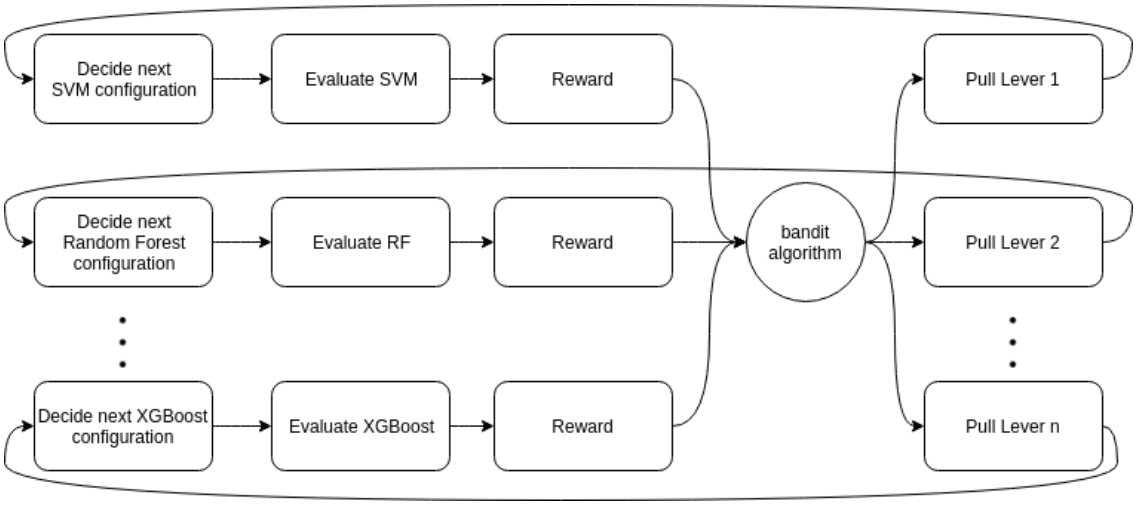


Figure 7.1: A visualization of our problem as a multi-armed bandit problem. The bandit algorithm decides which lever to pull based on the rewards it received from each lever.

7.2 Models and datasets

We use the flows 8315 (Random Forest), 8317 (Adaboost) and 6970 (SVM) and use our Blended Optimization technique as the hyperparameter optimization method. We run our experiments over 50 datasets and each meta-estimator gathers information from 50-82 datasets, where the current dataset is left out (leave-one-group-out).

7.3 Baselines

Since we experiment with the models Random Forest, AdaBoost and SVM, we use three baseline methods that always choose to optimize either Random Forest, AdaBoost or SVM. Furthermore, we add the random method, that randomly selects an algorithm to optimize.

7.4 Increasing effort

We create one extra method that we name "increasing effort". This method calculates the confidence interval for the observed scores so far on each lever or algorithm. If the upper limit of a limit exceeds the current optimum, we add this lever as a "candidate". Finally, we select the lever with the lowest number of "pulls".

Initially, we start with a confidence level of 90%. At some point it is possible that we find no candidates. In that case, we increase the confidence level to 95%. The next time we find no candidates, we increase it to 97.5%, and then to 98.75% and so forth. More formally, we calculate the confidence level as:

$$\texttt{CL} = 100 - 10/2^k\% \tag{7.1}$$

Where k is the number of times we ran out of candidates.

In Figure 7.2 we show the two steps that lead to selecting the final lever. In the first step, the algorithm tries to find candidate levers based on the 90% upper confidence bound of their observed scores. In this case, there are no candidates and the algorithm decides to use the 95% upper bound instead. It then finds two candidate levers, which upper confidence bound exceed the current optimum y^*. Then, in the second step, it selects the lever which, so far, had the least amount of iterations assigned to it.

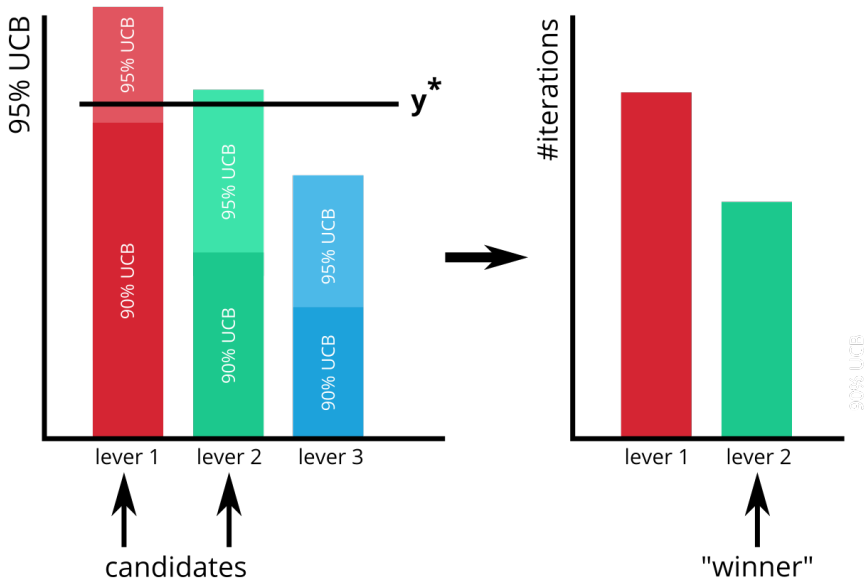


Figure 7.2: A visualization of increasing effort.

7.5 Results

We compare each algorithm, and if they require a parameter, we specify the set parameter next to the name. Furthermore, we add three important baselines: "always_the_best" repeatedly selects the algorithm that was initially the best, "always_randomforest" only selects the Random Forest algorithm, and "random" selects an algorithm purely at random.

Figure 7.3 shows that most methods do not improve upon simple randomized search. The methods `ucb1`, `ucb1_tuned`, `linear_decreasing_epsilon`, and our own `increasing_effort` do however surpass the randomized selection.

We think that our problem has one key difference over the normal multi-armed bandit problem that makes it hard for most algorithms to work well. Namely, in our problem, the goal is to find the algorithm with the highest optimum (i.e. the highest value for the 100th quantile), not the optimization process that finds better performing configurations on average, which is what most multi-armed bandit problems focus on.

The upper-bound methods might however still work reasonably, because they focus more on the possible upper limit that these rewards could have for each action. They are furthermore deterministic, which makes the optimization process easier to reproduce.

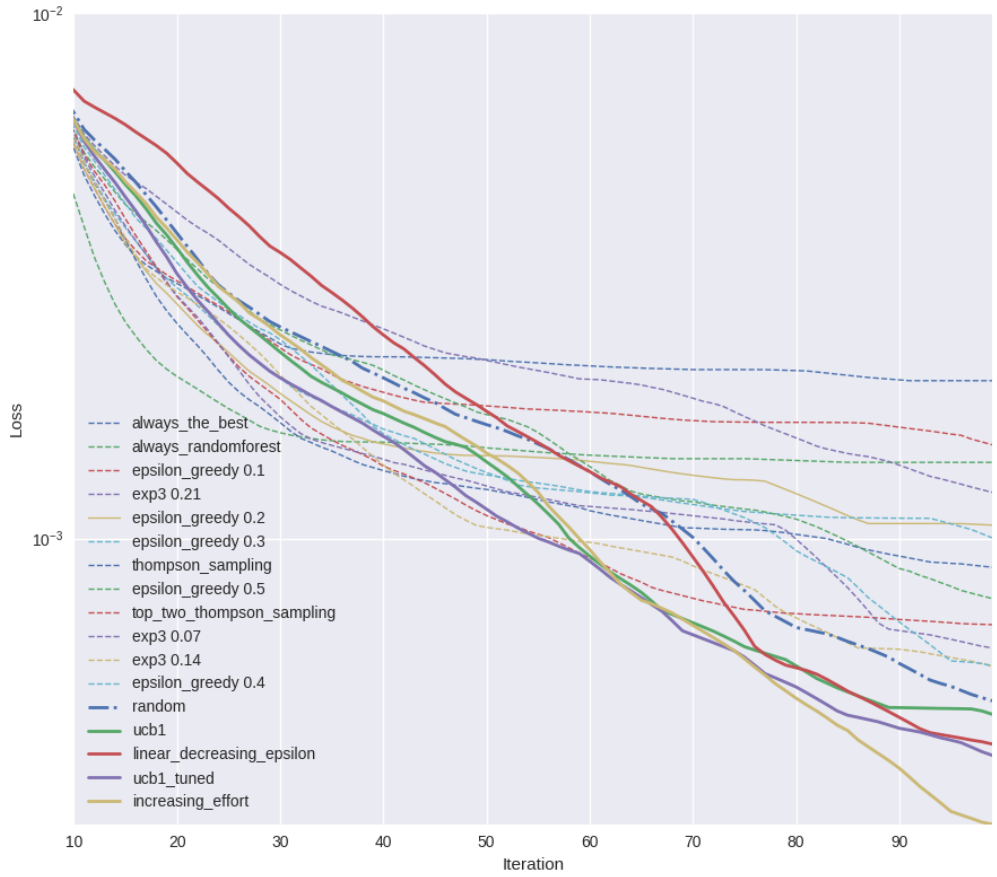


Figure 7.3: Comparison of multi-armed bandit algorithms on algorithm selection.

Chapter 8

Conclusions

We proposed a series of improvements on the current state-of-the-art automated machine learning tools. We will first cover our contributions, and then describe possible future work.

8.1 Contributions

We made the following contributions to the field of automated machine learning:

- **Hyperparameter optimization:** We proposed two new surrogate models based on LightGBM as a replacement for the Random Forest-based surrogate models in SMAC: one that models the performance and one that models the running time of each configuration. We furthermore proposed a new way of quantifying promising configurations by calculating an upper confidence bound using quantile regression. And we showed how this method could be modified so that we can include evaluation cost. We then showed that our proposed methods outperform the existing methods.

- **Meta-learning:** We proposed a new method to determine a set of configurations to warm-start hyperparameter optimization. Where AutoSklearn only saves the best-performing instantiation per dataset and uses KNN to select the most similar datasets, we train a meta-learner on all configurations executed on the OpenML platform. This meta-leaner learns to model the relative performance of configurations on different datasets, based on the workings of the hyperparameters, as well as the interaction between these configurations and the datasets' properties. We then introduced a simple formula that selects a warm-starting triple based on the meta-learner's predictions. We showed that our warm-starting method outperformed AutoSklearn's method.

- **Algorithm selection:** Instead of reducing the CASH problem to a hyperparameter optimization problem, we explored an alternative method, based on multi-armed bandits, for determining how much resources to allocate to optimizing each algorithm. We showed that most bandit algorithms would not work well because they try to find a lever that has the highest average reward. However, we also showed that a certain class of bandit algorithms, namely the ones that estimate an upper-bound, might be interesting for further research.

8.2 Future work

In this section we provide ideas that were left unexplored and alternatives that might improve the performance of our components even more.

8.2.1 Algorithm selection as a multi-armed bandit problem

We have briefly explored algorithm selection as a multi-armed bandit problem. We explained that there is a fundamental difference between our prlbem and the multi-armed bandit problem. We did however show that some class of algorithms gave promising results. As we have only tested our problem with a setup of a few machine learning algorithms, it might be interesting to experiment with a set of machine learning algorithms that is similar to what AutoSklearn uses, and then also compare these results against simply reducing the CASH problem to a hyperparameter optimization problem.

8.2.2 Hyperband with Bayesian Optimization

An interesting approach might be to combine Bayesian Optimization with Hyperband, which is shown to be successful by Falkner et al [8]. Hyperband is a multi-armed bandit strategy for hyperparameter optimization that uses so-called budgets b, which defines a cheap-to-evaluate approximate version $\tilde{f}(\cdot, b)$ of $f(\cdot)$. This budget can for example be the number of folds in a cross-validation to evaluate a specific hyperparameter configuration.

The original Hyperband repeatedly called SuccessiveHalving to identify the best out of n randomly sampled configurations. SuccessiveHalving sets a low budget, evaluates the sampled configurations with this low budget, and keeps only the best half. After that, it increases the budget for the better configurations, and so on. Instead, Falkner et al. use Bayesian optimization instead of randomly sampled configurations, and dub their framework BOHB (Bayesian Optimization + Hyperband).

The surrogate model here is trained on the beginning of each iteration, on all configurations that are evaluated on the largest budget. The surrogate model then tries to predict which configurations perform the best, and the SuccessiveHalving then takes these configurations instead of randomly sampled configurations.

As the optimization progresses, more configurations are evaluated on bigger budgets. Given that the goal is to optimize on the largest budget, BOHB always uses these configurations that are evaluated on the largest budget. This enables it to overcome wrong conclusions drawn on smaller budgets by eventually relying on results with the highest fidelity only.

8.2.3 SuccessiveHalving for algorithm selection

We can also use SuccessiveHalving for algorithm selection. This was used by AutoSklearn for their second AutoML challenge [1]. After running all of a set of ML pipelines for a small budget, it drops the worse half of the pipelines and doubles the budget of the remaining pipelines. This halving of pipelines and doubling of budget is successively repeated until only a single pipeline is trained on the whole budget.

8.2.4 Computing metafeatures

We have used the metafeatures that were pre-computed by OpenML. Instead, it would be interesting to remove metafeatures and investigate how well our meta-learner can still correlate after removing these metafeatures. If we compute these metafeatures ourselves, we can trade-off the duration it takes for these metafeatures to be calculated against the meta-learner's performance in ranking the configurations.

[1] https://www.automl.org/blog-2nd-automl-challenge/

Bibliography

[1] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2-3):235–256, 2002. 13, 17, 18

[2] Hilan Bensusan and Christophe Giraud-Carrier. Discovering task neighbourhoods through landmark learning performances. In *European Conference on Principles of Data Mining and Knowledge Discovery*, pages 325–330. Springer, 2000. 12

[3] Pavel Brazdil, João Gama, and Bob Henery. Characterizing the applicability of classification algorithms using meta-level learning. In *European conference on machine learning*, pages 83–102. Springer, 1994. 12

[4] Eric Brochu, Vlad M Cora, and Nando De Freitas. A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv preprint arXiv:1012.2599*, 2010. xi, 8, 37

[5] Rich Caruana, Alexandru Niculescu-Mizil, Geoff Crew, and Alex Ksikes. Ensemble selection from libraries of models. In *Proceedings of the twenty-first international conference on Machine learning*, page 18. ACM, 2004. 20

[6] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794. ACM, 2016. 22

[7] Katharina Eggensperger, Matthias Feurer, Frank Hutter, James Bergstra, Jasper Snoek, Holger Hoos, and Kevin Leyton-Brown. Towards an empirical foundation for assessing bayesian optimization of hyperparameters. In *NIPS workshop on Bayesian Optimization in Theory and Practice*, volume 10, page 3, 2013. 7

[8] Stefan Falkner, Aaron Klein, and Frank Hutter. Bohb: Robust and efficient hyperparameter optimization at scale. *arXiv preprint arXiv:1807.01774*, 2018. 72

[9] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 2962–2970. Curran Associates, Inc., 2015. 1, 2, 19, 20

[10] Jerome H Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001. 21

[11] Jerome H Friedman. Stochastic gradient boosting. *Computational Statistics & Data Analysis*, 38(4):367–378, 2002. 21

[12] Johannes Fürnkranz and Johann Petrak. An evaluation of landmarking variants. In *Working Notes of the ECML/PKDD 2000 Workshop on Integrating Aspects of Data Mining, Decision Support and Meta-Learning*, pages 57–68, 2001. 12

[13] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *International Conference on Learning and Intelligent Optimization*, pages 507–523. Springer, 2011. 1, 19, 20, 32, 33

[14] Frank Hutter, Lin Xu, Holger H Hoos, and Kevin Leyton-Brown. Algorithm runtime prediction: Methods & evaluation. *Artificial Intelligence*, 206:79–111, 2014. 32

[15] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. In *Advances in Neural Information Processing Systems*, pages 3146–3154, 2017. 21, 22

[16] Daniel James Lizotte. *Practical bayesian optimization*. University of Alberta, 2008. xi, 37, 38

[17] Randal S. Olson, Nathan Bartley, Ryan J. Urbanowicz, and Jason H. Moore. Evaluation of a tree-based pipeline optimization tool for automating data science. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, GECCO '16, pages 485–492, New York, NY, USA, 2016. ACM. 18

[18] Yonghong Peng1 Peter A Flach Pavel and Brazdil2 Carlos Soares. Decision tree-based data characterization for meta-learning. *IDDM-2002*, page 111, 2002. 13

[19] Bernhard Pfahringer, Hilan Bensusan, and Christophe G Giraud-Carrier. Meta-learning by landmarking various learning algorithms. In *ICML*, pages 743–750, 2000. 12

[20] KV Rashmi and Ran Gilad-Bachrach. Dart: Dropouts meet multiple additive regression trees. In *International Conference on Artificial Intelligence and Statistics*, pages 489–497, 2015. 21

[21] Daniel Russo. Simple bayesian algorithms for best arm identification. In *Conference on Learning Theory*, pages 1417–1418, 2016. 13, 17

[22] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 2951–2959. Curran Associates, Inc., 2012. 8

[23] Carlos Soares, Johann Petrak, and Pavel Brazdil. Sampling-based relative landmarks: Systematically test-driving algorithms before choosing. In *Portuguese conference on artificial intelligence*, pages 88–95. Springer, 2001. 12

[24] William R Thompson. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25(3/4):285–294, 1933. 13, 14

[25] Ricardo Vilalta, Christophe G Giraud-Carrier, Pavel Brazdil, and Carlos Soares. Using meta-learning to support data mining. *IJCSA*, 1(1):31–45, 2004. 12

[26] David H Wolpert. Stacked generalization. *Neural networks*, 5(2):241–259, 1992. 20

[27] Ciyou Zhu, Richard H Byrd, Peihuang Lu, and Jorge Nocedal. Algorithm 778: L-bfgs-b: Fortran subroutines for large-scale bound-constrained optimization. *ACM Transactions on Mathematical Software (TOMS)*, 23(4):550–560, 1997. 9

Appendix A

Preliminaries

We derive the closed-form expression of the Expected Improvement (EI) criterion commonly used in Bayesian Optimization.

Modelled with a Gaussian Process, the function value at a given point x can be considered as a normal random variable with mean μ and variance σ^2. Given the best (maximum in a maximization setup) function value obtained so far, which we will denote by f^*, we are interested in quantifying the improvement over f^* we will have if we sample a point x.

$$I(x) = \max(Y - f^*, 0) \tag{A.1}$$

where Y is the random variable $\sim \mathcal{N}(\mu, \sigma^2)$ that corresponds to the function value at x. Since I is a random variable, one can consider the average (expected) improvement (EI) to assess x:

$$EI(x) = E_{Y \sim \mathcal{N}(\mu, \sigma^2)}[I(x)] \tag{A.2}$$

With the reparameterization trick, $Y = \mu + \sigma\epsilon$ where $\epsilon \sim \mathcal{N}(0, 1)$, we have:

$$EI(x) = E_{\epsilon \sim \mathcal{N}(0,1)}[I(x)] \tag{A.3}$$

which can be written as (from linearity of integral, and the definition of $\frac{d}{d\epsilon}e^{-\epsilon^2/2}$ derivative)

$$EI(x) = \int_{-\infty}^{\infty} I(x)\phi(\epsilon)d\epsilon \tag{A.4}$$

$$EI(x) = \int_{-\infty}^{(\mu - f^*)/\sigma} (\mu + \sigma\epsilon - f^*)\phi(\epsilon)d\epsilon \tag{A.5}$$

$$EI(x) = (\mu - f^*)\Phi(\frac{\mu - f^*}{\sigma}) - \sigma \int_{-\infty}^{(\mu - f^*)/\sigma} \epsilon\phi(\epsilon)d\epsilon \tag{A.6}$$

$$EI(x) = (\mu - f^*)\Phi(\frac{\mu - f^*}{\sigma}) + \frac{\sigma}{\sqrt{2\pi}} \int_{-\infty}^{(\mu - f^*)/\sigma} (-\epsilon)e^{-\epsilon^2/2}d\epsilon \tag{A.7}$$

$$EI(x) = (\mu - f^*)\Phi(\frac{\mu - f^*}{\sigma}) + \frac{\sigma}{\sqrt{2\pi}} e^{-\epsilon^2/2}\Big|_{-\infty}^{(\mu - f^*)/\sigma} \tag{A.8}$$

$$EI(x) = (\mu - f^*)\Phi(\frac{\mu - f^*}{\sigma}) + \sigma\big(\phi(\frac{\mu - f^*}{\sigma}) - 0\big) \tag{A.9}$$

$$EI(x) = (\mu - f^*)\Phi(\frac{\mu - f^*}{\sigma}) + \sigma\phi(\frac{\mu - f^*}{\sigma}) \tag{A.10}$$

where ϕ, Φ are the PDF, CDF of standard normal distribution, respectively.

Appendix B

Meta learning

CfsSubsetEval_DecisionStumpAUC	Area Under the ROC Curve achieved by the landmarker weka.classifiers.trees.DecisionStump -E "weka.attributeSelection.CfsSubsetEval -P 1 -E 1" -S "weka.attributeSelection.BestFirst -D 1 -N 5" -W
CfsSubsetEval_DecisionStumpErrRate	Error rate achieved by the landmarker weka.classifiers.trees.DecisionStump -E "weka.attributeSelection.CfsSubsetEval -P 1 -E 1" -S "weka.attributeSelection.BestFirst -D 1 -N 5" -W
CfsSubsetEval_DecisionStumpKappa	Kappa coefficient achieved by the landmarker weka.classifiers.trees.DecisionStump -E "weka.attributeSelection.CfsSubsetEval -P 1 -E 1" -S "weka.attributeSelection.BestFirst -D 1 -N 5" -W
CfsSubsetEval_NaiveBayesAUC	Area Under the ROC Curve achieved by the landmarker weka.classifiers.bayes.NaiveBayes -E "weka.attributeSelection.CfsSubsetEval -P 1 -E 1" -S "weka.attributeSelection.BestFirst -D 1 -N 5" -W
CfsSubsetEval_NaiveBayesErrRate	Error rate achieved by the landmarker weka.classifiers.bayes.NaiveBayes -E "weka.attributeSelection.CfsSubsetEval -P 1 -E 1" -S "weka.attributeSelection.BestFirst -D 1 -N 5" -W
CfsSubsetEval_NaiveBayesKappa	Kappa coefficient achieved by the landmarker weka.classifiers.bayes.NaiveBayes -E "weka.attributeSelection.CfsSubsetEval -P 1 -E 1" -S "weka.attributeSelection.BestFirst -D 1 -N 5" -W
CfsSubsetEval_kNN1NAUC	Area Under the ROC Curve achieved by the landmarker weka.classifiers.lazy.IBk -E "weka.attributeSelection.CfsSubsetEval -P 1 -E 1" -S "weka.attributeSelection.BestFirst -D 1 -N 5" -W
CfsSubsetEval_kNN1NErrRate	Error rate achieved by the landmarker weka.classifiers.lazy.IBk -E "weka.attributeSelection.CfsSubsetEval -P 1 -E 1" -S "weka.attributeSelection.BestFirst -D 1 -N 5" -W

CfsSubsetEval_kNN1NKappa	Kappa coefficient achieved by the landmarker weka.classifiers.lazy.IBk -E "weka.attributeSelection.CfsSubsetEval -P 1 -E 1" -S "weka.attributeSelection.BestFirst -D 1 -N 5" -W
DecisionStumpAUC	Area Under the ROC Curve achieved by the landmarker weka.classifiers.trees.DecisionStump
DecisionStumpErrRate	Error rate achieved by the landmarker weka.classifiers.trees.DecisionStump
DecisionStumpKappa	Kappa coefficient achieved by the landmarker weka.classifiers.trees.DecisionStump
J48.00001.AUC	Area Under the ROC Curve achieved by the landmarker weka.classifiers.trees.J48 -C .00001
J48.00001.ErrRate	Error rate achieved by the landmarker weka.classifiers.trees.J48 -C .00001
J48.00001.Kappa	Kappa coefficient achieved by the landmarker weka.classifiers.trees.J48 -C .00001
J48.0001.AUC	Area Under the ROC Curve achieved by the landmarker weka.classifiers.trees.J48 -C .0001
J48.0001.ErrRate	Error rate achieved by the landmarker weka.classifiers.trees.J48 -C .0001
J48.0001.Kappa	Kappa coefficient achieved by the landmarker weka.classifiers.trees.J48 -C .0001
J48.001.AUC	Area Under the ROC Curve achieved by the landmarker weka.classifiers.trees.J48 -C .001
J48.001.ErrRate	Error rate achieved by the landmarker weka.classifiers.trees.J48 -C .001
J48.001.Kappa	Kappa coefficient achieved by the landmarker weka.classifiers.trees.J48 -C .001
NaiveBayesAUC	Area Under the ROC Curve achieved by the landmarker weka.classifiers.bayes.NaiveBayes
NaiveBayesErrRate	Error rate achieved by the landmarker weka.classifiers.bayes.NaiveBayes
NaiveBayesKappa	Kappa coefficient achieved by the landmarker weka.classifiers.bayes.NaiveBayes
REPTreeDepth1AUC	Area Under the ROC Curve achieved by the landmarker weka.classifiers.trees.REPTree -L 1
REPTreeDepth1ErrRate	Error rate achieved by the landmarker weka.classifiers.trees.REPTree -L 1
REPTreeDepth1Kappa	Kappa coefficient achieved by the landmarker weka.classifiers.trees.REPTree -L 1
REPTreeDepth2AUC	Area Under the ROC Curve achieved by the landmarker weka.classifiers.trees.REPTree -L 2
REPTreeDepth2ErrRate	Error rate achieved by the landmarker weka.classifiers.trees.REPTree -L 2
REPTreeDepth2Kappa	Kappa coefficient achieved by the landmarker weka.classifiers.trees.REPTree -L 2
REPTreeDepth3AUC	Area Under the ROC Curve achieved by the landmarker weka.classifiers.trees.REPTree -L 3
REPTreeDepth3ErrRate	Error rate achieved by the landmarker weka.classifiers.trees.REPTree -L 3

REPTreeDepth3Kappa	Kappa coefficient achieved by the landmarker weka.classifiers.trees.REPTree -L 3
RandomTreeDepth1AUC	Area Under the ROC Curve achieved by the landmarker weka.classifiers.trees.RandomTree -depth 1
RandomTreeDepth1ErrRate	Error rate achieved by the landmarker weka.classifiers.trees.RandomTree -depth 1
RandomTreeDepth1Kappa	Kappa coefficient achieved by the landmarker weka.classifiers.trees.RandomTree -depth 1
RandomTreeDepth2AUC	Area Under the ROC Curve achieved by the landmarker weka.classifiers.trees.RandomTree -depth 2
RandomTreeDepth2ErrRate	Error rate achieved by the landmarker weka.classifiers.trees.RandomTree -depth 2
RandomTreeDepth2Kappa	Kappa coefficient achieved by the landmarker weka.classifiers.trees.RandomTree -depth 2
RandomTreeDepth3AUC	Area Under the ROC Curve achieved by the landmarker weka.classifiers.trees.RandomTree -depth 3
RandomTreeDepth3ErrRate	Error rate achieved by the landmarker weka.classifiers.trees.RandomTree -depth 3
RandomTreeDepth3Kappa	Kappa coefficient achieved by the landmarker weka.classifiers.trees.RandomTree -depth 3
StdvNominalAttDistinctValues	Standard deviation of the number of distinct values among attributes of the nominal type.
kNN1NAUC	Area Under the ROC Curve achieved by the landmarker weka.classifiers.lazy.IBk
kNN1NErrRate	Error rate achieved by the landmarker weka.classifiers.lazy.IBk
kNN1NKappa	Kappa coefficient achieved by the landmarker weka.classifiers.lazy.IBk

Table B.1: List of landmarkers used for meta-learning.