DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Optimizing Hyperparameters in the SG++ Datamining Pipeline

Eric Johannes Koepke

DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Optimizing Hyperparameters in the SG++ Datamining Pipeline

Optimierung von Hyperparametern in der SG++ Datamining Pipeline

Author:	Eric Johannes Koepke
Supervisor:	Prof. Dr. Hans-Joachim Bungartz
Advisor:	Kilian Röhner, M.Sc.
	Paul Sarbu, M.Sc.
Submission Date:	April 16th 2018

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, April 16th 2018 Eric Johannes Koepke

Abstract

In Machine Learning there are often parameters of the model or the training algorithm that have to be known before the actual learning begins. These hyperparameters can make a big difference to the success of a machine learning model, especially since these models grow more complex as research on them advances. Advanced automatic hyperparameter optimization algorithms were developed to find optimal hyperparameters as fast as possible. I implement and compare two different approaches in the context of SG++, a toolbox that uses Sparse Grids to perform different classical machine learning tasks. Harmonica successively reduces the optimization search space while Bayesian Optimization tries the most promising hyperparameter setting based on previous results. I test them on regression and density estimation tasks and discuss the strengths and weaknesses of both to show different use cases. Harmonica requires more resources while being trivial to parallelize and more thorough in its search. Bayesian Optimization converges faster and finds the optimal solution as long as certain conditions are met.

Abstrakt

Viele Parameter für Machine Learning Modelle und Algorithmen müssen vor dem eigentlichen Lernprozess festgelegt werden. Diese sogenannten Hyperparameter können maßgeblich über den Erfolg eines Machine Learning Modells entscheiden, vor allem, da diese Modelle im Rahmen des wissenschaftlichen Fortschritts zunehmend komplexer werden. Fortgeschrittene automatische Hyperparameter-Optimierungsalgorithmen wurden entwickelt, um Hyperparameter so schnell wie möglich optimal zu besetzen. Ich implementiere und vergleiche zwei verschiedene Ansätze im Kontext von SG++, einer Software, die Sparse Grids benutzt, um verschiedene klassische Machine Learning Aufgaben zu bewältigen. Harmonica reduziert schrittweise den Suchraum der Optimierung, wogegen Bayes'sche Optimierung immer die vielversprechenste Hyperparameter-Konfiguration basierend auf vorherigen Resultaten testet. Ich überprüfe die Leistung der Algorithmen anhand von Regression und Dichte-Abschätzung in SG++, vergleiche sie und zeige verschiedene Anwendungsfälle. Harmonica benötigt mehr Resourcen, aber ist einfacher zu parallelisieren und gründlicher in der Suche. Bayes'sche Optimierung konvergiert schneller und findet das Optimum solang gewisse Bedingungen erfüllt sind.

Contents

1 Introduction

In this chapter I will introduce the general context in which hyperparameter optimization is important. In the end, the structure of the thesis will be outlined.

1.1 Machine Learning

In the past years, machine learning has seen a rise in popularity. Both a cause and a consequence of this is the usage of more computational power to support more complex models. In statistics and machine learning, data-based models are used to represent complex relations between different observable statistics. If these statistics are correlated, usually because of some presumed causality, the correlation can be used to predict some statistics based on the observation of others. A popular machine learning example is image classification. The statistics are the raw pixel data of an image and its class, describing the contents of the image in terms of predefined classes. Because the class membership is generally difficult to obtain algorithmically, a machine learning model can be used to predict it from the raw pixel data. To make this possible, the model needs data to "learn" from. This learning, also called training or fitting, is performed by an algorithm that modifies parameters of the model in such a way, that the model fits the data well afterwards. In the image classification example a neural network, the model, is trained by means of gradient descent, the algorithm, to minimize a loss function that models the quality of the fit. Step-wise the weights of the neurons in the neural network get adjusted until the loss function converges. After this learning process, the model has implicitly stored knowledge about the images in its parameters, which are the weights of its neurons. They enable the trained network to make correct predictions for future images.

1.2 Hyperparameters

As these models grow more flexible to be able to describe the growing amounts of data that are available, they also need more tuning and configuration to make use of that flexibility. This tuning often comes in the form of hyperparameters, a term that describes all parameters that have to be determined before the actual process of

fitting a model to the data can start. These hyperparameters exist because data-based models are designed to work in different scenarios, requiring modifications to both algorithm and model. In the past these modifications were often done by using domain knowledge (educated guessing) or rules of thumb. However, hyperparameters are generally difficult to set. A common hyperparameter in the neural network case is the learning rate. It changes the rate at which neuron weights are adjusted per learning step and is essential for the performance of a neural network. While the consensus is that low learning rates slow learning down and high learning rates might keep the network from converging, the best choice depends on the data, because knowing the best learning approach beforehand essentially requires to have solved the initial problem beforehand as well.

To choose hyperparameters effectively and automatically, a range of hyperparameter optimization approaches exist, most of them train the model multiple times with different hyperparameter settings. For my thesis I implemented and tested two algorithms and applied them to machine learning tasks performed with SG++, a toolbox using sparse grids for a range of different applications.

1.3 Thesis Outline

In Chapter 2, I will introduce hyperparameter optimization, past work in the domain and important notation. I will then give a brief introduction into SG++ in Chapter 3. To gain an in-depth understanding I recommend reading [Pfl10]. After that, in Chapter 4 I will explain the two algorithms Harmonica and Bayesian Optimization with a focus on general strategy and mathematical theory. Implementational details follow in Chapter 5, where I discuss the structure of the SG++ datamining pipeline, my additions to it and specifics regarding both algorithms. In the end, in Chapter 6 I present my results and in Chapter 7 I draw conclusions and present opportunities for future work.

2 Hyperparamter Optimization

In this chapter, I am going to explain hyperparameters in the context of optimization and refer to existing literature on the topic.

Hyperparameters in machine learning describe a set of variables that modify how a certain model is derived from data. These parameters can modify the algorithm that performs this process but they can also be a parameter of the model that the algorithm can't reasonably determine itself. Most model parameters are being determined through training by applying the machine learning algorithm to the data. Typically efficient methods like gradient descent or analytical minimization are used to optimize these parameters during training. By exploiting the model structure and smoothness assumptions, these optimization techniques can be fast and precise. For hyperparameters this process is more difficult because the optimization problem is far less constrained. Hyperparameters are usually not independent of each other so the number of possible combinations of hyperparameter values rise exponentially with the amount of hyperparameters. In the past, hyperparameters were set by the machine learning practitioner based on domain knowledge and trial and error. However, as machine learning models got more complex and sensitive to hyperparameter choices, the use of automatic hyperparameter optimization methods became more important.

Because training machine learning models is computationally expensive, the main goal is to find good or optimal points with as few function evaluations as possible. The most trivial approach to solve this problem is random search. This means that for each evaluation all hyperparameters are chosen randomly within their viable range. Although this is not very efficient, there are several advantages to random search over more sophisticated algorithms. Besides the ease of implementation, it's also trivial to parallelize and most notably it needs no assumptions about the function or the underlying search space to be valid.

In the end every efficient hyperparameter optimization algorithm makes some assumptions and exploits them to gain an advantage over random search. Choosing the best algorithm therefore depends a lot on the characteristics of the model and the kind of hyperparameters it has.

2.1 Previous Work

During my thesis I mainly focused on two algorithms: Harmonica [HKY17] and Bayesian Optimization [SLA12][BCF10][Sha+16]. The first uses random search as a baseline but reduces the search space after it has gathered enough information. The second algorithm models the function as a Bayesian Process and maximizes an acquisition function based on all previous results to determine the best point to sample next.

Other approaches include Hyperband [Li+16] and Fabolas [Kle+17], both of which use dynamic resource allocation. Hyperband is an advanced form of successive halving [JT15], a technique that tests a lot of samples with low resources and grants more resources to promising samples. These resources can be training episodes or dataset sizes. Fabolas is a Bayesian Optimization variant that samples across variable dataset size while optimizing the information about the optimum at full dataset size. This can save time while being able to test more samples. In some domains, where gradients are available, hypergradients can be calculated by chaining gradients through the training process [MDA15].

2.2 Parameters

I have identified three different types of parameter. The main types are continuous and discrete parameters. Optimization over them is different in several aspects and the algorithms use different tools to deal with them. In terms of implementation it's also reasonable to use different number types to store their values. The third parameter type, the categorical parameter, is a variant of the discrete parameter. The difference lies in the spacial relationship of different values. Discrete parameters are assumed to be ordered while categorical are not. For example, 2 is a middle point between 1 and 3, however it might be difficult to find such a middle point for a number of different functions like linear, exponential and sine. For Bayesian Optimization this differentiation is important because it uses distance to reason about similarity. Harmonica doesn't do this so it is not affected. My solution is to set the distance between two categories of a categorical parameter to 1 if they are not the same and 0 if they are. Geometrically this is equal to placing n different categories on the corners of a regular n-simplex in an $(n-1)$-dimensional space. For two, three, and four categories that would be a line, triangle, and tetrahedron respectively. The geometrical explanation only gives the guarantee that this is indeed a valid distance measure in Euclidean space, the implementation $d(p_1, p_2) = (p_1 != p_2)$ of course doesn't rely on it.

2.3 Evaluation Function

Now I am going to take a quick look at the evaluation process in the context of optimization and clarify notation. This is about the process of fitting or training a model to train data and then testing it on test data (or performing cross-validation). It can be seen as a function mapping from the space of hyperparameters to a score. I will call the function evaluation function and the input a hyperparameter configuration or simply a sample point. The output corresponds to the mean squared error between prediction and ground truth when applying the trained model to the test set.

This function can now be studied with respect to optimization options. In general, the function is assumed to be non-convex as convexity would require the hyperparameters to interact in a specific way. In fact with a good number of hyperparameters it's highly likely that multiple local optima exist. Looking at the smoothness of the function, noise is important. In the presence of noise, the function is generally not smooth, which is why typically, there are no gradient-based methods found in hyperparameter optimization. When accounting for noise, the underlying function can still be smooth. This becomes important when choosing the kernel function for Bayesian Optimization later in Section 4.2.2.

2.4 Exploration vs. Exploitation

Exploration and exploitation are two important counterparts when describing search strategies. The idea is that every algorithm has to choose between exploring the domain further in the hopes of finding a new separate previously unknown optimum and exploiting the current optimum to find an even better point nearby. This trade-off in the end decides about speed and success at finding the true optimum. An exploitation heavy algorithm will usually be faster with a chance of finding the true optimum very late or not at all. Random Search is a good example of a purely exploration based method. It has no mechanism to exploit the knowledge gained while sampling. From the two algorithms I implement, Bayesian Optimization has a stronger focus on exploitation, while Harmonica spends a lot of time on exploration. These terms will be used throughout my thesis, most importantly when discussing results in Chapter 6.

3 SG++

SG++ is a toolbox implemented in C++ that provides various methods using adaptive sparse grids as a mathematical model. These methods range from solving differential equations over likelihood approximation to datamining. In the datamining context sparse grids are used to approximate functions for tasks like regression or classification on big datasets. Recently there have been efforts to create a user-friendly pipeline that makes many of the already existing algorithms accessible through a single program that can be configured with a json file. Over the course of my Bachelor's thesis I explored different techniques for hyperparameter optimization while implementing a module for this task for the datamining pipeline. In this chapter, I explain sparse grids and the hyperparameters I optimize.

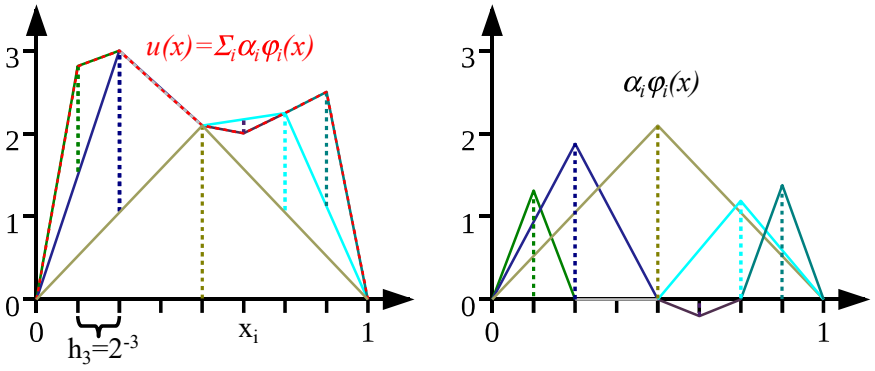

Figure 3.1: Composition of the red function left by adding of the hierarchical "hat" basis functions on the right. (from [Pfl10])

3.1 Sparse Grids

Sparse grids is a numerical method to represent a multidimensional function based on a discretization in the form of a grid. Usually grid-based approaches require a number of grid points that grows exponentially with the number of dimensions. This makes computations quickly too expansive to perform reasonably, a phenomenon known as the curse of dimensionality. Sparse grids provide a way of overcoming this to some extent by only modeling important grid points while leaving out the majority of the full grid. This is done by placing hierarchical basis functions on the grid and combining them to build an approximation of the modeled function (see Figure 3.1). By placing basis functions hierarchically as seen on the right and determining the right combination of coefficients, the red function on the left can be constructed. This technique allows the leaving out of a lot of basis functions whose coefficients would be very small. [Pfl10, Chapter 2]

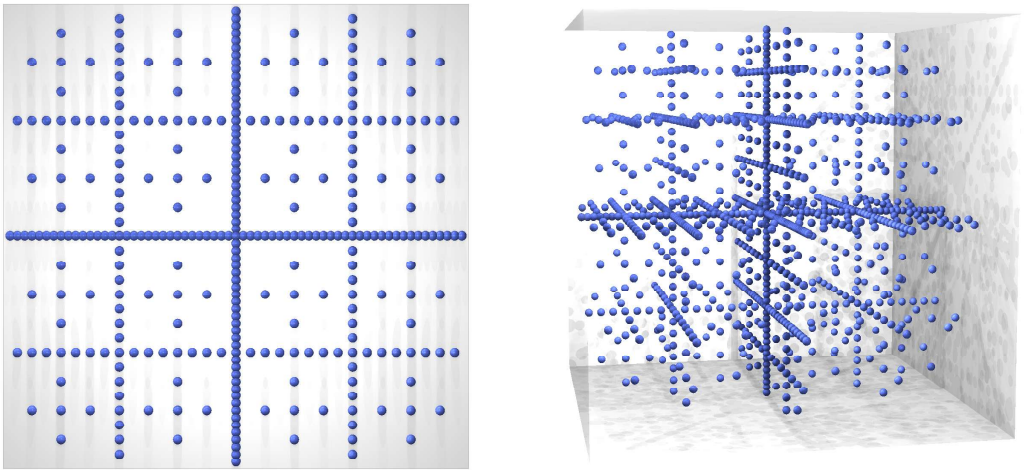

Figure 3.2: Unrefined grid structure of level 6 for two (left) and three (right) dimensions. (from [Pfl10])

3.2 Refinements

Sparse grids can be created a priori in certain structures that manage to approximate a function efficiently (see Figure 3.2). It has been shown by [BG04] that, under certain smoothness assumptions, the approximation has relatively low bounds in computation cost as well as in error. However, regardless of smoothness, local error estimation on

the grid can be used to refine it by adding grid points in regions, where the function approximation is the worst. This is done after a grid is fitted to data and such local error estimation becomes possible. There are different refinement criteria possible, the easiest being the coefficients of the existing basis functions. High coefficients indicate big deviations from the parent basis function in the hierarchical structure. If a grid point is identified as a good refinement candidate, its hierarchical children are added to the grid. Multiple points can be refined at once this way during one refinement step. After that the grid point coefficients get reevaluated. Multiple of such refinement steps can be performed. [Pfl10, Chapter 2]

3.3 Parameters

Having covered the basics of sparse grids, I will list and explain the different hyperparameters present in the datamining pipeline. I optimize hyperparameters for two applications, regression and density estimation. The last is a part of the bigger classification task. Both applications use sparse grids in a similar way and rely on the same hyperparameters governing grid construction. When discussing test results in Chapter 6, I will explain technical differences. For now, I will treat them equally.

One very important hyperparameter fixes the number of refinement steps that are performed. Including this parameter in the optimization poses a problem though. Because refinement steps are performed in order on the same existing model, testing multiple times with a different maximum number of refinement steps leads to doing redundant calculations. There are different possible workarounds for this. After evaluating a sample, the results for every refinement step could be returned. This would however complicate the basic definition of the optimization problem and every algorithm would need some specific tweak to deal with that. Additionally if the algorithm later wanted to test more refinements it would still have to recalculate all previous refinement steps. Because of these problems I decided to leave the parameter out of the optimization and instead refine every model as long as it improves the score up to a maximum number of refinements that has to be fixed beforehand. I do this because the decline in performance after a certain amount of refinements can be attributed to overfitting and some experimental runs have shown it to be unlikely that performance increases again after falling once.

There are generally a number of different basis functions available, I will use two of those: Linear and modified linear. Both are "hat" functions as shown in Figure 3.1 for the 1-dimensional case. On the inside of the grid they are identical but on the boundary, the simple approach is always 0. The modified linear basis function solves this problem by going up at the boundary instead of down.

Table 3.1: Hyperparameters for SG++ regression

Name	Type
Grid Basis Function	Categorical
Grid Level	Discrete
Max. Refinement Points	Discrete
Refinement Threshold	Continuous
Regularization Lambda	Continuous

The grid level is the hierarchical depth of the unrefined grid in its standard pattern as visible in Figure 3.2. The pattern is already sparse so it doesn't contain all possible grid points but the number of points still rise exponentially with the grid level.

The refinement threshold defines the minimum value for the refinement criterion necessary for refinement.

The maximum number of refinement points refers to the points that are selected for refinement. The limit doesn't have to be exhausted but it can be beneficial to have such a limit.

Lambda is a regularization parameter that affects the process of evaluating the coefficients for the basis functions. A higher lambda generally leads to smaller coefficients. This can be used to counter overfitting but when to high will drastically flatten the function approximation.

4 Algorithms

This chapter is about the two hyperparameter optimization algorithms I tested and implemented. I will describe the procedure of each algorithm while roughly explaining important maths along the way. Some steps will be explained in more detail in the implementation chapter. At the end of the chapter, I will briefly talk about parellelization as it is important to compare both algorithms.

4.1 Harmonica

The first algorithm, called Harmonica, was recently introduced in [HKY17]. For an in-depth explanation, I recommend reading the original paper as I will only explain core aspects and skip over a lot of theory.

Harmonica uses knowledge derived from Fourier Analysis of Boolean functions to reduce the search space through multiple stages. Essentially it applies regression on the evaluation function after doing a sufficient number of (random) samples. The regression is done with special predictor variables that allow the algorithm to restrict the search space based on the regression results. This means that Harmonica looks for general trends in the function hoping that the optimum can be found following that trend. This works especially well if there are a lot of hyperparameters and if the interaction between them is not too complicated on a bigger level.

4.1.1 Parameter Space

The algorithm expects a hyperparameter configuration to consist of only boolean variables so an implementation needs to provide a transformation to boolean space. For continuous hyperparameters this means that the parameter has to be reduced to a discrete space with 2^{b_i} values resulting in b_i boolean variables that represent the parameter in the algorithm. For discrete variables I recommend to test a range of different values equal to a power of 2. Otherwise the mapping to boolean space either has to map values twice or leave some out completely which could potentially lead to undesired results. For clarification, harmonica has no concept of the hyperparameters that are represented by the boolean variables. It can only bring them into context by means of the constraints explained below.

4.1.2 Regression

Once the mapping is complete, n (around 200) samples are randomly drawn from the space of 2^b total combinations of boolean parameters. The drawing process could also be done non-random but, to preserve theoretical guarantees, it is important that the values of each parameter are expected to appear equally. After all samples are evaluated, the boolean search space gets halved successively by introducing constraints based on regression coefficients. The predictor variables are generated by the parity function $\{-1,1\}^n \Rightarrow \{-1,1\} : p(X) = \prod_{x \in X} x$. Sets of up to 3 boolean parameters are used to create predictor variables that model the interaction of these parameters. Sets of size 1 are simply all the parameters themselves. Sets of size 2 are all combinations of two parameters. The predictors calculated from these sets will be 1 for samples that have the same (boolean) values for both boolean parameters included in the calculation of that predictor. If the boolean parameters have different values, they are -1. This is useful because it enables the algorithm to model any situation where the choice of two parameters depends on each other. If, for example, two boolean parameters are used to represent the same continuous parameter, their joined predictor separates the two central points on the scale from the two outer points. Predictors for sets of size 3 are calculated accordingly and can model more complicated relationships.

Table 4.1: Example for joined predictor with continuous parameters.

Continuous Parameter	0	1	2	3
Boolean Parameters	$(-1,-1)$	$(-1,1)$	$(1,-1)$	$(1,1)$
Joined Predictor	1	-1	-1	1

Ordinary Least Squares Regression with an L_1-Regularization term, so-called Lasso Regularization, generates coefficients for each predictor variable. Sorted by absolute value the highest coefficients are transformed into constraints such that the predictor variable can only take on the better value in the constrained search space.

4.1.3 Search space reduction

This means that after adding a number of constraints, some boolean variables are either fixed in place or (if the predictor contained multiple variables) dependent on other variables. The independent remaining variables can then be used to construct the constrained space. In the next stage there are again n (or less) random samples drawn, now from the constrained space. This is done by randomly setting the independent

variables and adjusting the others accordingly. After the samples are evaluated, another round of constraints can be added and so on.

4.2 Bayesian Optimization

Bayesian Optimization is a relatively old procedure, first described in [Moc77]. The general strategy of Bayesian Optimization is to view the evaluation function as a random function, place a prior over it and use the posterior to predict the performance of future samples. The random function is typically modeled with a Gaussian Process. This means that the sample points are jointly Gaussian distributed. To learn more about Gaussian Processes I recommend reading [RW06]. I will only highlight the mathematical properties that are relevant for implementation. After that I will talk about the choice of the kernel function, a measure of similarity used in the Gaussian Process. Finally the acquisition function takes the Gaussian Process prediction and returns a measure for the potential that lies in sampling that point next. By maximizing this function the next sample point is determined. This point is then evaluated and the Gaussian process in updated which in turn changes the acquisition function. The procedure is visualized in Figure 4.1.

4.2.1 Gaussian Process

Since the Gaussian Process describes a multivariate normal distribution over all samples, a single sample point can be described by a univariate Gaussian with statistics $\mu_t(x)$ and $\sigma_t^2(x)$ [BCF10]. Intuitively speaking, points that are close to an already sampled point have a similar mean and low variance. Those farther away have high variance and a mean that relies mostly on the prior mean. Using those two values, with the acquisition function one can pursue both exploration (high variance) and exploitation (high mean). Assuming there are already a number of observations $\{\mathbf{x}_{1:t}, \mathbf{f}_{1:t}\}$, mean and variance of a new candidate x_{t+1} can be calculated as follows [BCF10]:

$$\mathbf{K} = \begin{bmatrix} k(x_1, x_1) & \dots & k(x_1, x_t) \\ \vdots & \ddots & \vdots \\ k(x_t, x_1) & \dots & k(x_t, x_t) \end{bmatrix} \tag{4.1}$$

The covariance matrix $\mathbf{K}$ of the Gaussian Process is a Gram matrix (or kernel matrix) built by the kernel function $k(x, x')$. As long as the kernel function $k(x, x')$ is a valid kernel [RW06], this matrix will always be symmetric and positive-definite. This property can later be of use in the implementation section 5.3.

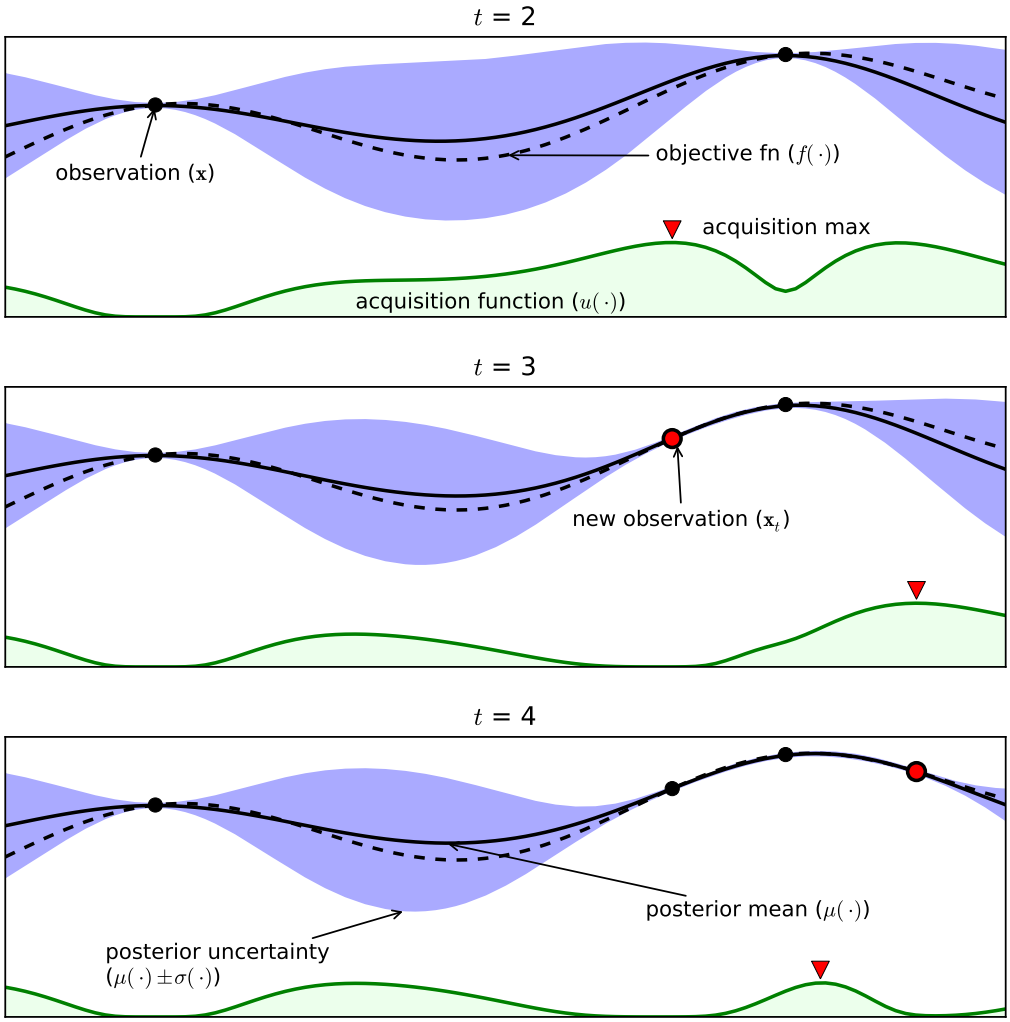

Figure 4.1: Demonstration of Bayesian Optimization on a 1D toy problem. Multiple steps show the change in Gaussian Process predictions and acquisition function. (from [BCF10])

Additionally for a new candidate x_{t+1} I need the similarity to the observations as given by [BCF10]:

$$\mathbf{k} = \begin{bmatrix} k(x_{t+1}, x_1) \\ \vdots \\ k(x_{t+1}, x_t) \end{bmatrix} \tag{4.2}$$

Now for this candidate the mean and variance are [BCF10]:

$$\mu_t(x_{t+1}) = \mathbf{k}^T \mathbf{K}^{-1} \mathbf{f}_{1:t} \tag{4.3}$$

$$\sigma_t^2(x_{t+1}) = k(x_{t+1}, x_{t+1}) - \mathbf{k}^T \mathbf{K}^{-1} \mathbf{k} \tag{4.4}$$

4.2.2 Kernel Function

As just shown, the purpose of the kernel function is to model the covariance matrix of the Gaussian Process by providing a similarity measure. As such it is essentially defining the main properties of the model function. Whether it can approximate the evaluation function well depends on the choice of the kernel. The most common kernel is the squared exponential kernel:

$$k(x, x') = e^{-\frac{\|x - x'\|^2}{2\sigma^2}} \tag{4.5}$$

There are also other options that differ mostly in smoothness. The squared exponential kernel is infinitely differentiable so it's a smooth kernel. Testing different kernel functions could be beneficial in the future but for now only one kernel was used. Now there is still an open parameter σ^2 that is different from the σ_t^2 used in the Gaussian Process. This variance measure decides how detailed the function model will be. A smaller σ^2 will cause points with equal distance to be perceived as less similar. This increases the detail as points are less influenced by the global trend. However with the mention of distance there arises a bigger problem. To calculate this distance the different hyperparameters have to be put in a spacial relation of some kind. The σ^2 can then be seen as a scaling factor for that space. Since introducing additional hyperparameters in hyperparameter optimization is always unwanted there are methods to find the scaling factors for each hyperparameter. This process is called Automatic Relevance Determination (ARD) because the scaling factor can be seen as the relevance of a parameter for the output of the evaluation function. The simplest approach here is to maximize the marginal likelihood of the Gaussian Process fit under different kernel parameters, which simplifies to [Sha+16]:

$$p(\mathbf{f}_{1:t}|\mathbf{x}_{1:t}) \sim -\mathbf{f}_{1:t}^T\mathbf{K}^{-1}\mathbf{f}_{1:t} - log|\mathbf{K}| \tag{4.6}$$

The two parts of that likelihood can be interpreted as model fit and model complexity as smoother matrices have smaller determinants [Sha+16]. This means fitting a Gaussian Process behaves similar to logistic regression.

4.2.3 Acquisition Function

The acquisition function combines mean and variance given by the Gaussian Process in a sampling quality measure. There are several different acquisition functions to choose from. The simplest calculates the probability of improvement over the best previous result. On first thought this is a good idea but it doesn't take into account the amount of improvement that is achieved and thus focuses on minimal improvements. The next step is to calculate expected improvement [BCF10]:

$$EI(x) = \begin{cases} (\mu(x) - f(x^+))\Phi(Z) + \sigma(x)\phi(Z) & \text{if } \sigma(x) > 0 \\ 0 & \text{if } \sigma(x) = 0 \end{cases} \tag{4.7}$$

$$Z = \frac{\mu(x) - f(x^+)}{\sigma(x)} \tag{4.8}$$

It calculates the expectation over the deviation of the mean that would result in an overall improvement. Now the potential of uncertain points to outperform the best previous result (x^+) by a bigger margin is captured as well. There are other acquisition functions like Entropy Search, which tries to predict the information gain about the optimum but a comparison in [SLA12] shows that Expected Improvement can compete with more recent approaches while still being relatively simple to implement.

4.3 Parallelization

When solving problems with high computational demand as is the case in hyperparameter optimization, parallelization can often help to accelerate testing cycles. This is why, even though I did not implement parallel routines, it is important to compare the algorithms with respect to potential parellelization. Bayesian Optimization is a mathematically very elegant way of achieving good results in a short amount of time. However, it is inherently serial and therefore difficult to parallelize. Still there are multiple approaches for parallel Bayesian Optimization. One is building a joined acquisition function for the next 4 or 8 samples, but this makes the auxiliary optimization a

lot more difficult [WF16]. Another pretends that the result for a running evaluation is already in and starts another evaluation based on the predicted value. All these approaches manage to use parallel architecture but the increase in performance is usually not proportional to the additional resources spent. Harmonica, on the other hand, always evaluates a lot of samples at once so it is trivial to parallelize and able to fully use additional resources. This means that every performance comparison depends on the amount of resources available.

5 Implementation

In this chapter I will explain the class structure of the datamining pipeline in SG++ with focus on my own additions. Afterwards implementational details for specific parts of both algorithms will be given, procedures and terms introduced in Chapter 4 are assumed to be known.

5.1 Datamining Pipeline

Here I will give an overview over the structure of the code I worked on. In Figure 5.1 I provide a simple class diagram to visualize the relationship of the new classes.

5.1.1 Module Structure

At the core of the datamining pipeline are three modules that are implemented using the "Abstract Factory Pattern". The **DataSource** provides access to the data, the **Fitter** contains the model like regression or classification and the **Scorer** handles interaction between the two, managing evaluation and scoring of the model based on test sets or cross-validation.

5.1.2 New Module: HPO

My addition to the project is the hpo module. It contains classes that are directly responsible for the implementation of the algorithms, hyperparameter representation classes and **FitterFactory** classes that build **Fitter** classes with their corresponding configuration and provide a model-independent interface for the algorithms. The abstract **FitterFactory** stores maps for each of the three parameter types that map strings to the respective parameter class. It has methods that allow the algorithms to get information about these parameters and then manipulate them. The concrete factory adds parameters to those maps in it's constructor and implements a *buildFitter()* method that assembles a configuration for the specific fitter by getting (manipulated) values from the parameter classes.

The hyperparameter classes contain the range on which the parameter is supposed to be tested as well as discretization choices specific to Harmonica. **HyperparameterOpti-**

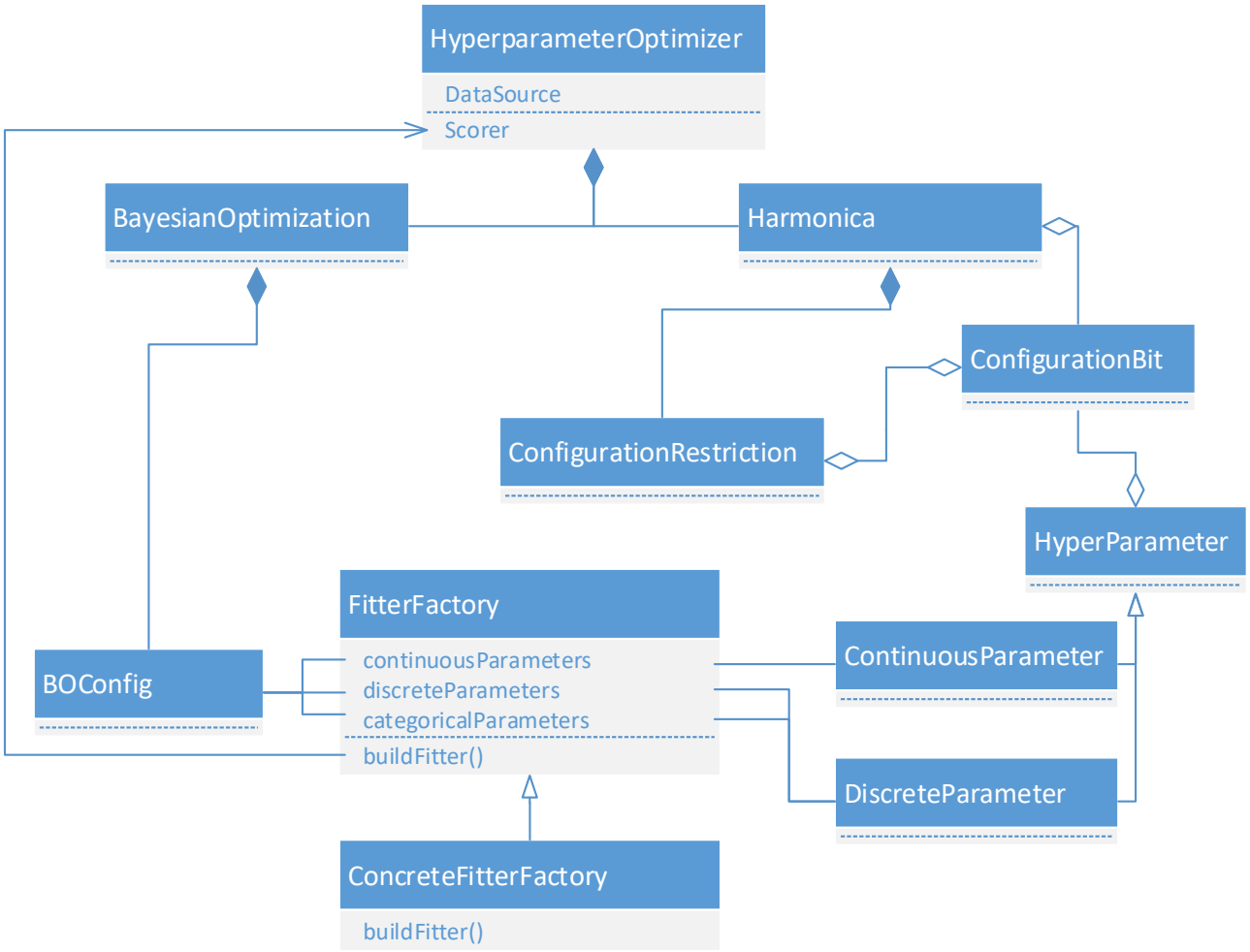

Figure 5.1: Module Structure. Manipulation of hyperparameters is realized through abstract factory and parameter classes.

mizer is the class that similar to **SparseGridMiner** in the base package brings together the different modules, in this case the **DataSource**, the concrete **FitterFactory** and the **HPOScorer** which is a modification of the **SplittingScorer**. The **HyperparameterOptimizer** is assembled by a **UniversalMinerFactory** that calls configuration processing and creates all the important classes.

For both of the two algorithms Harmonica and Bayesian Optimization there is a main class that provides all main methods to run the algorithm in multiple steps. They are called from the the **HyperparameterOptimizer**. There are also a range of helper classes to represent different mechanics used in each algorithm. Additionally both algorithms require functionality from the Solver and Optimization modules in SG++.

5.2 Harmonica

This section explains how the different steps explained in Section 4.1 are implemented and what classes are used to represent the different structures.

5.2.1 Preparation

Starting with Harmonica the implementation is centered around the **ConfigurationBit** class that represents the boolean variables Harmonica takes as input. The objects of this class are stored by the parameter classes and pointers to them are retrieved through the **FitterFactory**. Now the first step in the algorithm is to create random configurations. This is done by creating a list of random integers without duplicates. They are then translated into the boolean representation $\{-1, 1\}^b$ and used to set the **ConfigurationBits** accordingly. Simultaneously the **Fitters** are build and the regression matrix is filled. A structure of pointers to the **ConfigurationBits** is used to model which bits each predictor variable is calculated from. This structure is later reused to build the constraints.

5.2.2 Model Evaluation

After the preparation step the **Fitters** are evaluated. This is currently done in a for-loop so parallel implementation on one or even multiple machines would be possible without much intrusion into the system. The main part, calculating the constrained space, follows next.

5.2.3 Regression

The Lasso Regression is performed using the results and the previously build matrix. This can be done with an SLE-solver. I used the **Fista** solver from the Solver module because it already has the Lasso Regularization function as an option. Because the solver is build for sparse grids but the matrix is relatively small I had to provide standard matrix multiplication without usage of a grid to the solver in a helper class. After the regression is done, I sort the weight vector by absolute value and add constraints based on the sign of the weight.

5.2.4 Constraint Generation

For each constraint an object of class **ConfigurationRestriction** is created that is attached to the **ConfigurationBits** that are affected. After that the method *fixConfigBits()* is called for the first time. It's general purpose is to resolve all existing constraints. It does so by searching for constraints for which all but one bit are already set. The value of that last bit can then be calculated. Once there are no more constraints that can be resolved, any bit can be set without causing a contradiction. In my case this will be the first unset bit that will at the same time be added to a list of free bits. This list will later be used to create configurations in the reduced space. Right now, while adding constraints, after resolving is finished, all of the constraints should be satisfied. If this is not the case, the new constraint is removed again as it is in conflict with already existing constraints.

5.2.5 Stages

Once the desired amount of constraints is added the previously taken samples are reconsidered. If they lie in the constrained space, their information is still useful for the next stage and can be used again in the next regression phase. The next stage starts again by creating a list of unique random integers that get converted to boolean space. This time the reduced space $\{-1, 1\}^{b-r}$ gets translated into the full space by resolving the constraints for each configuration using the *fixConfigBits()* function. After that the fitters are again evaluated and new constraints built. This process can be repeated as often as desired but in practice 2-3 stages are a good compromise between exploration and exploitation. [HKY17]

5.3 Bayesian Optimization

This section explains details mostly surrounding numerics and optimization for Bayesian Optimization. In practice to start the algorithm, the Gaussian Process is initially built with a few (here 10) random samples. Hyperparameter configurations for Bayesian Optimization are stored by a dedicated class that also provides a range of helper methods. Then the implementation mostly follows the procedure described in Section 4.2.

5.3.1 Matrix Decomposition

First the kernel matrix is formed. Calculating mean and variance requires the inverse, but matrix inversion can lead to numerical instability. However, because a kernel matrix is always positive-definite (see Section 4.2.1), Cholesky Decomposition can be applied. I supplied two methods for decomposing and then solving a system accordingly. The decomposition only has to be done once per sample while the solving is done for every sample candidate during optimization of the acquisition function.

5.3.2 Acquisition Optimization

For this optimization I iterate over all combinations of discrete parameters and for every combination I run a gradient-free solver to optimize the continuous parameters. I use a **Mulistart** Nelder-Mead algorithm provided by the optimization module of SG++. It starts at multiple random points and iteratively optimizes from there. It expects a function mapping a vector to a scalar. This function is provided in the form of the *acquisitionOuter()* function that handles calculation of kernel values and ultimately calls the desired acquisition function. (here: Expected Improvement) It is to be noted that the time required to perform this auxiliary optimization is assumed to be insignificant compared to evaluating the score function. This assumption holds true in all tested cases but in general the optimization method might have to be tuned accordingly by reducing the number of start points or the number of optimization steps.

After the optimization is done, the best configuration gets evaluated and the result is used to update the Gaussian Process. Another row/column is added to the (symmetrical) kernel matrix and it is decomposed again.

5.3.3 ARD

Another important step is to perform the Automatic Relevance Determination (ARD). For this I reused the same optimizer as for the acquisition function though gradient-based solvers or more advanced ARD methods could be used. Because ARD modifies the kernel for each optimization step, the kernel matrix has to be decomposed for every

step as well. The time required lies in $O(n^3)$ steps, where n is the number of samples, so ARD is significantly more expensive than maximizing the acquisition function. For my case it was still viable to do ARD after every sample but this is not strictly necessary.

6 Results

This chapter is about testing the algorithms in the SG++ context and discussing strengths and weaknesses. I will introduce the different test cases and afterwards explain and interpret the results visible in the figures.

6.1 Test Cases

Most of my testing focuses on the regression task in the datamining pipeline. This is because density estimation was only partly available in the pipeline at the time of testing. Specifically refinement was not possible for density estimation, which is responsible for most of the hyperparameters.

6.1.1 Regression

For regression I tested on two different datasets that are available through the SG++ project and were used before in [Pfl10] for regression testing.

Table 6.1: Hyperparameters for SG++ regression

Name	Type	Range	Bits
Grid Basis Function	Categorical	linear, modlinear	1
Grid Level	Discrete	$[1, 4]$	2
Max. Refinement Points	Discrete	$[1, 4]$	2
Refinement Threshold	Continuous	$10^{[-5,-2]}$	3
Regularization Lambda	Continuous	$10^{[-7,0]}$	5

The first is the artificial 10-dimensional friedman dataset. Five of those dimensions are correlated with the target while the rest serve as decoy variables. Additionally there is a normally distributed noise term with variance 1 so the best MSE achievable without overfitting is also given by 1. The second dataset is the DR5 dataset which is a real-world dataset used for redshift estimation of galaxies. It has four dimensions and contains mostly data with high observation certainty. For both datasets the training set size is set to 10,000 while the test set is 50,000 in size. Testing is relatively fast and

there is enough data available, which allows getting high accuracy in performance evaluation.

The hyperparameters for regression contain all parameters introduced in Chapter 3. The ranges are based on prior tests but could be applied to most datasets as they are relatively broad.

6.1.2 Density Estimation

For density estimation I optimized two hyperparameters (grid level and regularization lambda) when running density estimation on a handcrafted dataset. For this task I designed a 4-dimensional probability density function like this:

$$p(a,b,c,d) = (3-6|a-0.5|)^2(-6b^2+6b)(2a+2)(5b+2)c^{2a+1}(1-c^{2a+2})^{5b+1} \quad (6.1)$$

The function, visualized in Figure 6.1, is a valid PDF on the interval $[0,1]$ with absolute density values under 10 making it relatively broad. It consists of two quadratic components a and b and a Kumaraswamy distribution for c that takes a and b as parameters to allow for interdimensional interactions. d doesn't affect the density at all, so the function is uniformly distributed in d. It's important to notice that this PDF is close to 0 at all boundaries and mostly smooth with the exception of the seam caused by the absolute in a. I wrote a simple script to create datasets that are distributed according to this function. To follow the implementation of the pipeline, testing is done by calculating the mean squared error between density estimation and actual density for a test set that itself is distributed according to the PDF. The procedure could be altered to test the estimation at different points yielding different results. However, I think accuracy in the high density ranges might be more important for tasks like classification, so I stayed with that testing procedure. Since training without refining is fast, both train and test sets contain 50,000 samples each.

Table 6.2: Hyperparameters for SG++ density estimation

Name	Type	Range	Bits
Grid Level	Discrete	$[4,7]$	2
Regularization Lambda	Continuous	$10^{[-10,0]}$	7

The hyperparameters for density estimation testing are grid level and the lambda from regularization. I wanted to optimize over the basis function as well but the modified linear basis function didn't work for high grid levels producing numerical

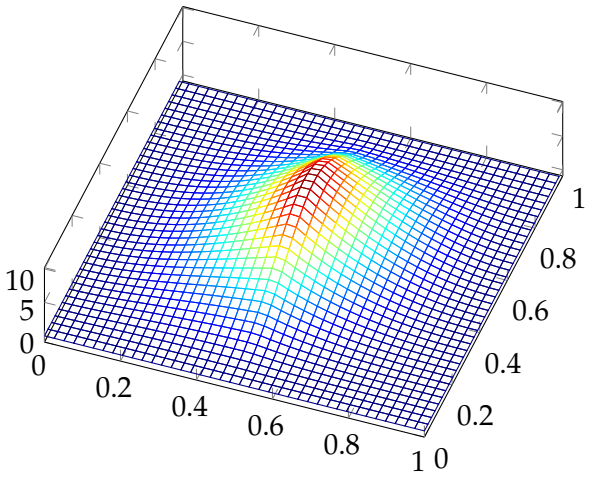

Figure 6.1: Artificial probability density function on domain $[0,1]^4$ for $c = 0.5$ and any value for d.

instability errors. As it performs worse on lower levels as well, I decided to fix the basis function to linear only. Because refinement was unavailable and no grid points can be added after the initial fit, higher grid levels are needed to get good results. This is why I only tested higher grid levels.

6.2 Test Results

Here are the results of my test and a few words of interpretation for each. Overall, the results show that both algorithms work well for all test scenarios.

6.2.1 Friedman

In Figure 6.2 you can see the performance of the two algorithms on the Friedman dataset. For clarity of the graph I don't show the individual sample points.

First off, there appear to be hyperparameter configurations that don't really work in the sense that the error is near 200 for quite a number of samples. For this reason I cut the graph in three parts that show different parts of the error range, all linearly scaled. This way I can demonstrate different things in the same graph. For both algorithms I show the samples that improve over the previous best result, as well as the median, which is calculated for a block of 25 samples (one column in the graph). The median is a good indicator for the general search strategy. Harmonica searches completely random in its first stage before applying two constraints. These constraints manage to

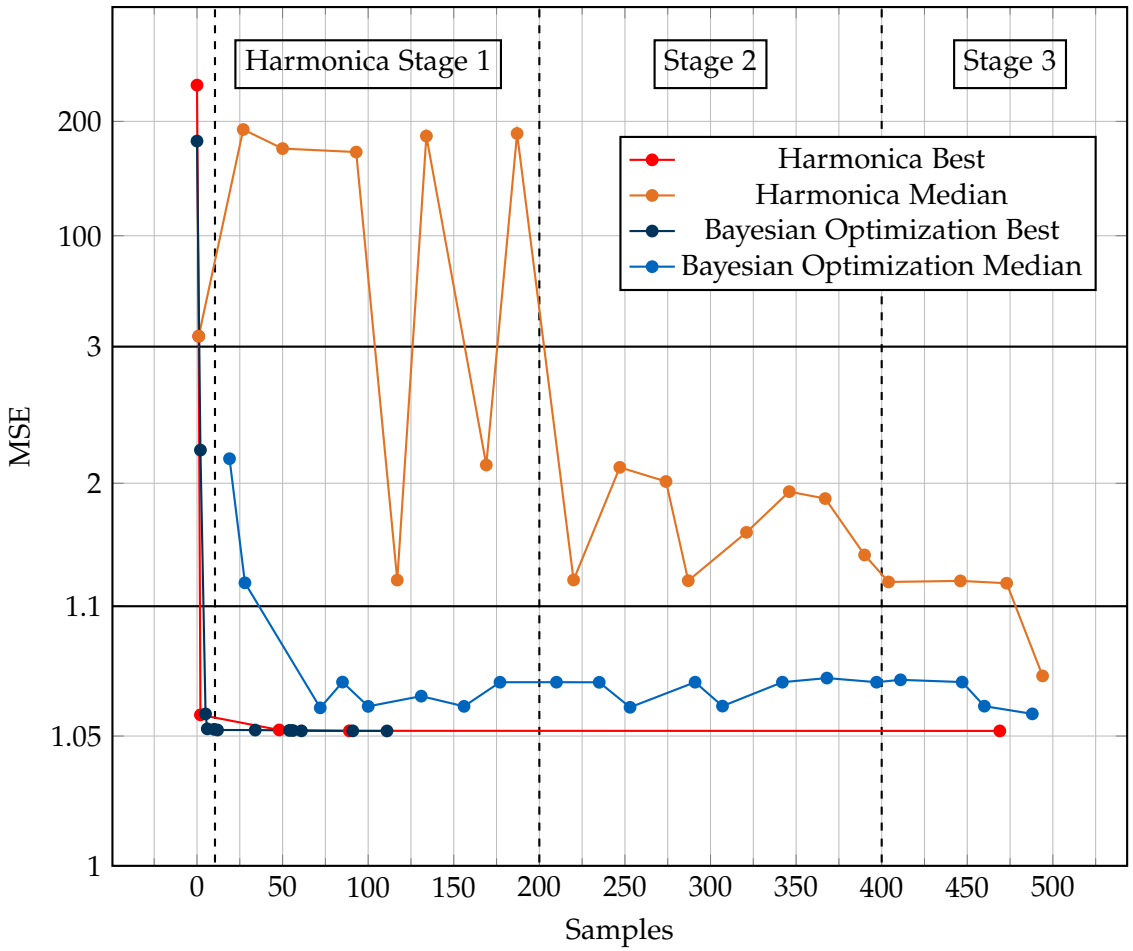

Figure 6.2: Test error over time on Friedman dataset, scaled linearly to highlight three different ranges. Median is evaluated for 25 samples each and demonstrates search behaviour. Bayesian Optimization focuses on good regions quickly while Harmonica does so slowly with each stage. Both algorithms find points near the optimum fast.

exclude the worst samples so the median drops to the medium range between 1 and 3. In its final stage after applying two more constraints, the median drops again and it can improve on its previous best result by a very small amount. Bayesian Optimization on the other hand explores mostly during the first 75 samples. Then the median is constantly very close to the optimum showing its greedy search strategy. Note that Bayesian Optimization still explores the rest of the search space using a small fraction

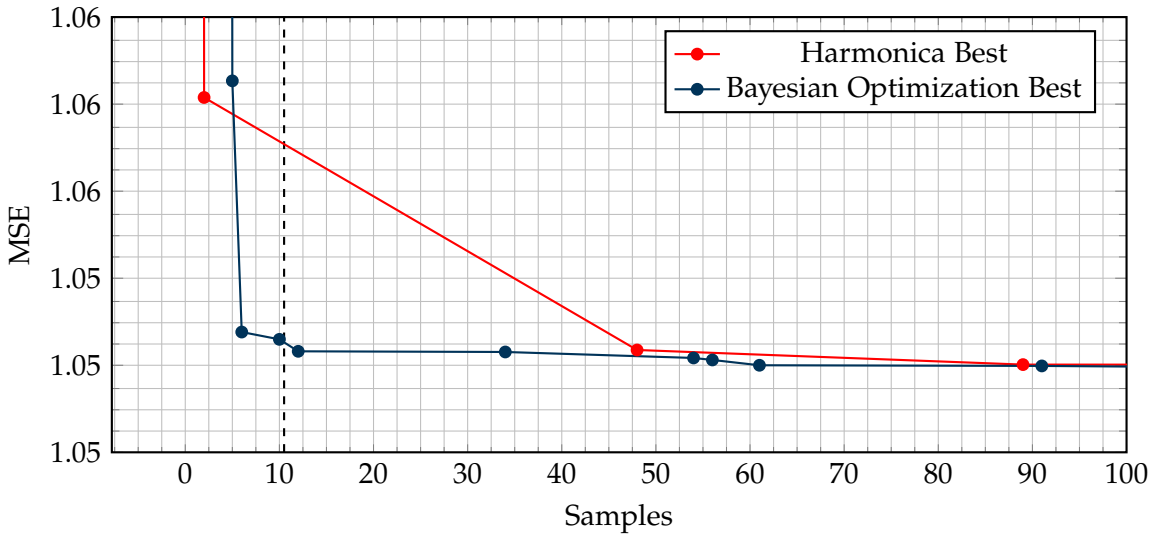

Figure 6.3: Test error over time on Friedman datset, enhanced to show initial development. Dashed line marks end of random search for Bayesian Optimization. It improves more frequently than Harmonica, which is de facto random search in stage 1.

of its time, this can not be seen by looking at the median. Most importantly, both algorithms find the optimum within approximately 100 samples. A closeup of this region can be seen in Figure 6.3. The first 10 samples for Bayesian Optimization are random sampled as a warm up. At this point it has already found some good results. This means that the optimum for this task and dataset is relatively wide and easy to find. However, Bayesian Optimization can still improve frequently, especially comparing to Harmonica that is completely random during those first 100 samples.

6.2.2 DR5

The test on the DR5 dataset, visible in Figure 6.4, produces results, differing in a few aspects. Overall the results, plotted to base 10^{-3} are very good and comparable to results shown in [Pfl10]. However, in this test Bayesian Optimization fails to find the same optimum that Harmonica finds. After inspecting the data, there appears to be a small sub-optimum, that the grid points of Harmonica happen to fall into. This doesn't mean, Harmonica is the better algorithm but it demonstrates potential problems of both algorithms. Bayesian Optimization doesn't always explore enough and Harmonica is dependent on its grid discretization.

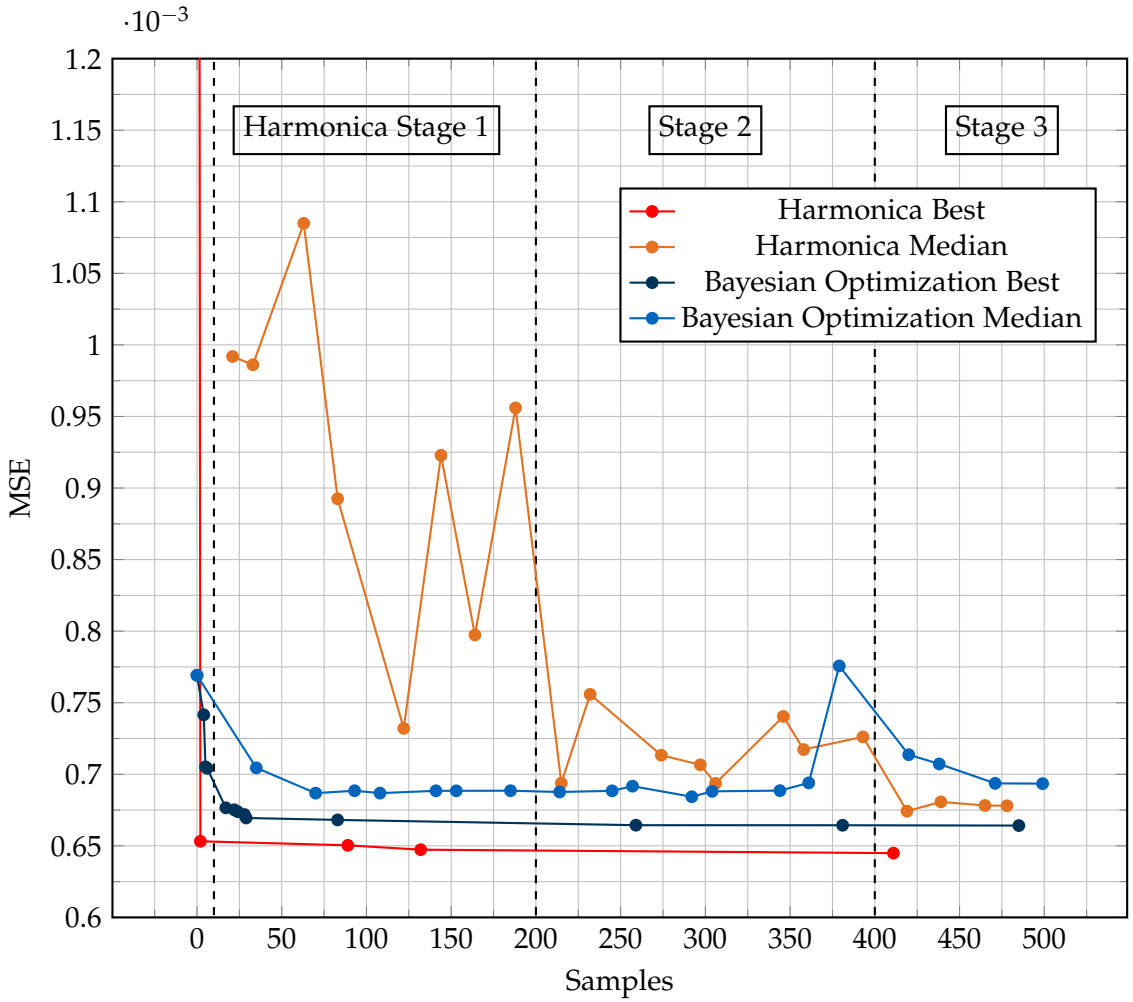

Figure 6.4: Test error over time on DR5 dataset. Bayesian Optimization fails to find the true optimum, but Harmonica might have a lucky discretization advantage.

6.2.3 Density Estimation

Because there are less hyperparameters involved the search is generally easier and harmonica can be run with less samples per stage but, apart from that, the results are similar to the other tests. Compared to the other tests, the error is relatively high as a result of missing refinement.

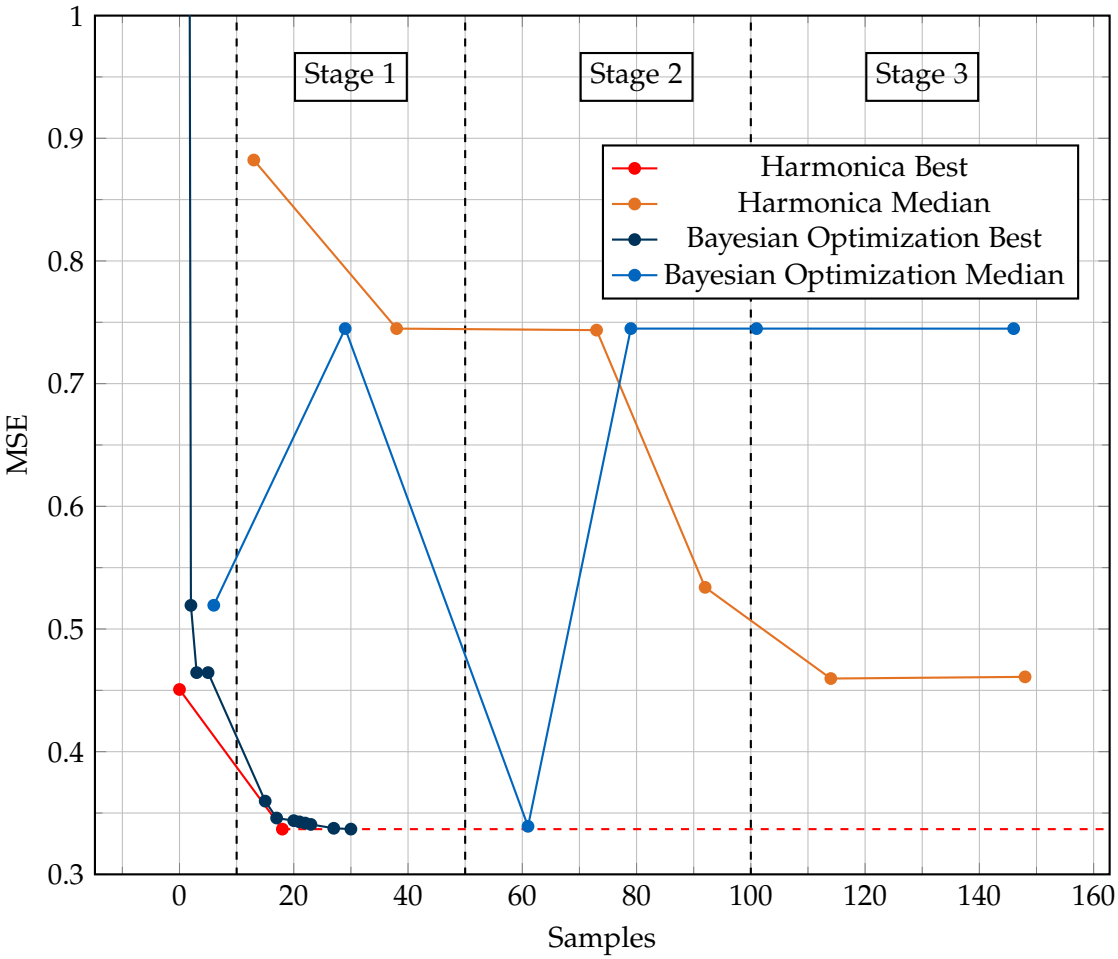

Figure 6.5: Test error over time on handcrafted density estimation dataset. Because of low dimensionality in the hyperparameters, both algorithms perform similar. The median is mostly governed by the discrete choice of grid level as small changes in the continuous lambda don't affect the result a lot.

7 Conclusions

Hyperparameter optimization is a complicated topic and after research, implementation and testing I can say that there is no algorithm that is right for every situation. For application in the SG++ datamining pipeline, Bayesian Optimization has proven to deliver good results fast, making it a good standard tool to make learning from data work without requiring prior knowledge about any hyperparameters. Harmonica is probably better suited for use in multi-machine parallel settings. It needs a lot of samples but explores a big part of the search space and could handle a lot more parameters than currently needed for SG++ [HKY17].

7.1 Future Work

In the end I will present a few possible topics for future work. All of those suggestions should fit in with my current implementation and improve upon the current state in a meaningful way.

7.1.1 Parallelization

As explained in Chapter 4, Harmonica is easy to implement in parallel although cross-machine parellelization would still require significant setup. In my opinion this kind of resource intensive optimization is only useful for very specific situations but it can deliver certainty that indeed the optimum is found. Bayesian Optimization is significantly more difficult to parallelize and requires adopting different approaches on a fundamental level. Considering that a lot of the evaluation computation in SG++ can already be run in parallel, I expect there to be only minimal gain from a parallel Bayesian Optimization approach.

7.1.2 Dataset Subsampling

An interesting approach that could work for SG++ is dataset subsampling. Basically it means that samples are taken with different amounts of data in the training set. The idea is to save time by doing cheaper samples on smaller training sets while still learning about the behaviour at full size. The biggest problem is, that this correlation is

previously unknown. Hyperband [Li+16] circumvents this by sampling across a wide range of sizes and running the best samples of each size bracket on the full dataset. This approach might work for SG++ but the base principle can also be applied to Bayesian Optimization and, to some extent, Harmonica. Fabolas [Kle+17] is a version of Bayesian Optimization that optimizes both the evaluation function and the evaluation time needed to learn as quickly as possible about the optimum at full dataset size. And according to the authors, Harmonica can learn constraints on small dataset subsamples as well, but this has the risk of excluding the optimum based on false presumptions about the underlying correlations. From small test it seems that good results on small datasets often transfer to good results on large datasets for regression in SG++ but theoretically this can't be guaranteed.

7.1.3 Combining Algorithms

Another possibility is to combine different approaches. For example, after exploring the search space with harmonica, the results could be used as input for Bayesian Optimization to refine them further. This way the different strengths of both algorithms can be combined.

Bibliography

[BCF10] E. Brochu, V. M. Cora, and N. de Freitas. "A Tutorial on Bayesian Optimization of Expensive Cost Functions, with Application to Active User Modeling and Hierarchical Reinforcement Learning." In: *CoRR* abs/1012.2599 (2010). arXiv: 1012.2599. URL: http://arxiv.org/abs/1012.2599.

[BG04] H.-J. Bungartz and M. Griebel. "Sparse grids." In: *Acta Numerica* 13 (2004), pp. 147–269. DOI: 10.1017/S0962492904000182.

[HKY17] E. Hazan, A. R. Klivans, and Y. Yuan. "Hyperparameter Optimization: A Spectral Approach." In: *CoRR* abs/1706.00764 (2017). arXiv: 1706.00764. URL: http://arxiv.org/abs/1706.00764.

[JT15] K. G. Jamieson and A. Talwalkar. "Non-stochastic Best Arm Identification and Hyperparameter Optimization." In: *CoRR* abs/1502.07943 (2015). arXiv: 1502.07943. URL: http://arxiv.org/abs/1502.07943.

[Kle+17] A. Klein, S. Falkner, S. Bartels, P. Hennig, and F. Hutter. "Fast Bayesian Optimization of Machine Learning Hyperparameters on Large Datasets." In: *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics, AISTATS 2017, 20-22 April 2017, Fort Lauderdale, FL, USA*. 2017, pp. 528–536. URL: http://proceedings.mlr.press/v54/klein17a.html.

[Li+16] L. Li, K. G. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar. "Efficient Hyperparameter Optimization and Infinitely Many Armed Bandits." In: *CoRR* abs/1603.06560 (2016). arXiv: 1603.06560. URL: http://arxiv.org/abs/1603.06560.

[MDA15] D. Maclaurin, D. K. Duvenaud, and R. P. Adams. "Gradient-based Hyperparameter Optimization through Reversible Learning." In: *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*. 2015, pp. 2113–2122. URL: http://jmlr.org/proceedings/papers/v37/maclaurin15.html.

[Moc77] J. Mockus. "On Bayesian Methods for Seeking the Extremum and their Application." In: *IFIP Congress*. 1977, pp. 195–200.

[Pfl10] D. Pflüger. "Spatially Adaptive Sparse Grids for High-Dimensional Problems." PhD thesis. Technical University Munich, 2010. ISBN: 978-3-86853-555-6. URL: http://www.dr.hut-verlag.de/978-3-86853-555-6.html.

[RW06] C. E. Rasmussen and C. K. I. Williams. *Gaussian processes for machine learning*. Adaptive computation and machine learning. MIT Press, 2006. ISBN: 026218253X. URL: http://www.worldcat.org/oclc/61285753.

[Sha+16] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. de Freitas. "Taking the Human Out of the Loop: A Review of Bayesian Optimization." In: *Proceedings of the IEEE* 104.1 (2016), pp. 148–175. DOI: 10.1109/JPROC.2015.2494218. URL: https://doi.org/10.1109/JPROC.2015.2494218.

[SLA12] J. Snoek, H. Larochelle, and R. P. Adams. "Practical Bayesian Optimization of Machine Learning Algorithms." In: *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States.* 2012, pp. 2960–2968. URL: http://papers.nips.cc/paper/4522-practical-bayesian-optimization-of-machine-learning-algorithms.

[WF16] J. Wu and P. I. Frazier. "The Parallel Knowledge Gradient Method for Batch Bayesian Optimization." In: *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*. 2016, pp. 3126–3134. URL: http://papers.nips.cc/paper/6307-the-parallel-knowledge-gradient-method-for-batch-bayesian-optimization.