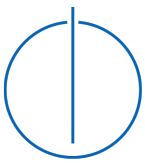
DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics: Games Engineering

Hyperparameter Optimization for Machine Learning Applications with the Sparse Grid Density Estimation

Sonja Doppelfeld



DEPARTMENT OF INFORMATICS

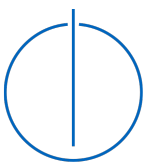
TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics: Games Engineering

Hyperparameter Optimization for Machine Learning Applications with the Sparse Grid Density Estimation

Hyperparameteroptimierung für Anwendungen des maschinellen Lernens mit der Dünngitter Dichteschätzung

Author:	Sonja Doppelfeld
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	M.Sc. Michael Obersteiner
Submission Date:	September 15, 2021



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

Munich, September 13, 2021 Sonja Doppelfeld

Acknowledgements

I want to acknowledge

- ..my advisor, Michael Obersteiner, for his help and patience,
- ..my friends and family who have supported me by sharing their expertise, offering their time and brightening my days,
- ..everyone and everything else that has helped to keep up my mood.

Abstract

Due to the increasing amount of data, machine learning algorithms have gained importance, as they automatically process large data sets. These algorithms are calibrated by hyper-parameters, that need to be chosen carefully. Hyper-parameter optimization methods are used to automatize this process. This thesis discusses hyper-parameter optimization in the context of sparse grid density estimation. First, the concept of density estimation is introduced, followed by classification and clustering, two common types of machine learning that can be based on it. Afterwards, hyper-parameters are defined, and various methods to optimize them are presented. Grid search, random search, and Bayesian optimization are discussed in theory and in the context of my implementation. These methods are used to optimize the hyper-parameters for normal and adaptive classification. Finally, the performance of the implemented methods is analyzed and compared to that of the open source software hyperopt.

Zusammenfassung

Durch die zunehmende Menge an Daten haben Algorithmen für maschinelles Lernen an Relevanz gewonnen, da diese automatisch große Datensätze verarbeiten. Diese Algorithmen werden von Hyperparametern kalibriert, die sorgfältig ausgewählt werden müssen. Hyperparameteroptimierungsmethoden werden verwendet um diesen Prozess zu automatisieren. Diese Arbeit bespricht Hyperparameteroptimierung im Kontext von Dünngitter Dichteschätzung. Zuerst wird das Konzept der Dichteschätzung eingeführt, gefolgt von Klassifizierung und Clustering, zwei verbreiteten Arten von maschinellem Lernen, die darauf basieren können. Danach werden Hyperparameter definiert, sowie diverse Methoden präsentiert diese zu optimieren. Rastersuche, Zufallssuche sowie Bayessche Optimierung werden theoretisch und im Kontext meiner Implementierung besprochen. Diese Methoden werden verwendet um die Hyperparameter von normaler und adaptiver Klassifizierung zu optimieren. Die Leistung der implementierten Methoden wird im letzten Schritt analysiert, sowie mit der Leistung der Open-Source-Software hyperopt verglichen.

Contents

1 Introduction

In recent years, the amount of data has increased drastically, up to a point where it is often impossible to manually process it. To handle this huge amount of data, methods are needed that automatically analyze data. *Machine learning* methods have the goal of finding patterns in a given data set. These patterns are further used "to predict future data or other outcomes of interest" [1], such as analyzing a new data point with them. *Classification* and *clustering* are two common machine learning methods. The former is given a set of labeled data, meaning that each data point belongs to a specified "class". It then tries to find patterns in this data to understand the structure of these classes, so that it can then classify unseen data. Clustering is similar, but the given data is not labeled. Therefore the classes are not specified beforehand, but the algorithm tries to find out which data points may be put together. These groups of data points are then called "clusters" [1]. Both can be implemented with the help of *density estimation*, a method that reconstructs the unknown density function of a given data set.

For machine learning methods, there are usually parameters that need to be specified before the learning process, called *hyper-parameters*. They significantly influence the training accuracy and speed of the method, and therefore should be carefully configured [2]. This raises the question that this thesis aims to answer: "How can the hyper-parameters of a given machine learning algorithm be optimized?". In particular, I will discuss this question in the context of sparse grid density estimation.

In Chapter 2, an introduction to density estimation is given, as well as further explanations on classification and clustering. Chapter 3 then provides a definition of hyper-parameters and presents various hyper-parameter optimization methods, focused on the method that I have implemented in the scope of this thesis. This implementation is then explained in Chapter 4. Chapter 5 discusses tests that have been conducted on it, as well as their results and lastly, the thesis is concluded by Chapter 6 providing a review and outlook.

2 Density Estimation and Machine Learning

This Chapter introduces density estimation, a tool that estimates the underlying density function of a given data set, in Section 2.1. Classification, a machine learning algorithm that learns the structure of classes of a set of labeled data, is then explained in Section 2.2. Clustering, an algorithm that works on unlabeled data points and groups them into clusters, is then discussed in Section 2.3.

2.1 Density Estimation

This section is mainly based on [3], if not stated otherwise. Density estimation is a tool used to reconstruct the underlying density function of a given data set, which can also provide a basis for data-mining tasks like clustering and classification. It is given a data set $S = \{x_1, \ldots, x_M\} \subset \mathbb{R}^d$, which consists of samples drawn from an unknown distribution with a probability density function f. The task is then to estimate f based on the given data S, constructing the estimated density function $\hat{f}$ of f. There are two types of density estimation methods: *Parametric density estimation* requires additional information about the underlying distribution, while *nonparametric density estimation* uses only the data set S [3].

2.1.1 Kernel Based Density Estimation

A common nonparametric method is *Kernel density estimation*, which uses a kernel function, often the *Gaussian kernel*:

$$\hat{f}(x) = \frac{1}{M} \sum_{i=1}^{M} K(\frac{x - x_i}{h}), \tag{2.1a}$$

$$K(x) = (2\pi)^{-1/2} e^{-x^2/2}, \tag{2.1b}$$

with $h > 0$ being the bandwidth. Selecting a good value for h is a non-trivial problem. Kernel density estimators can also become very expensive for large datasets, because in order to evaluate $\hat{f}$ as many kernels as there are data points in S need to be computed [3].

2.1.2 Sparse Grid Density Estimation

Density estimations based on *sparse grids* improves both these problems: The idea of grid-based density estimation is to first estimate the density function using a highly overfitted guess f_ϵ, and to then obtain a smoother, more generalized approximation $\hat{f}$ using *spline smoothing*. Instead of using kernels, the density function $\hat{f}$ is discretized using basis functions centered at grid points, and does not need one basis function for each data point. This improves the computational cost as long as the amount of grid points is considerably smaller

than the amount of data points, but it still suffers from the *curse of dimensionality* when used with regular grids. This means that, for a *full grid*, the amount of grid points grows exponentially with the dimension of the data points. To improve this, sparse grids are introduced [4]. In particular, the *sparse grid combination technique* can be used for this problem [5]. In this method, the solution is calculated independently on anisotropic and therefore inexpensive full grids. These results are then combined to obtain the final solution. Figure 2.1 shows an example for the construction of a density function using a sparse grid, Figure 2.2 provides a visualization of the sparse grid combination technique.

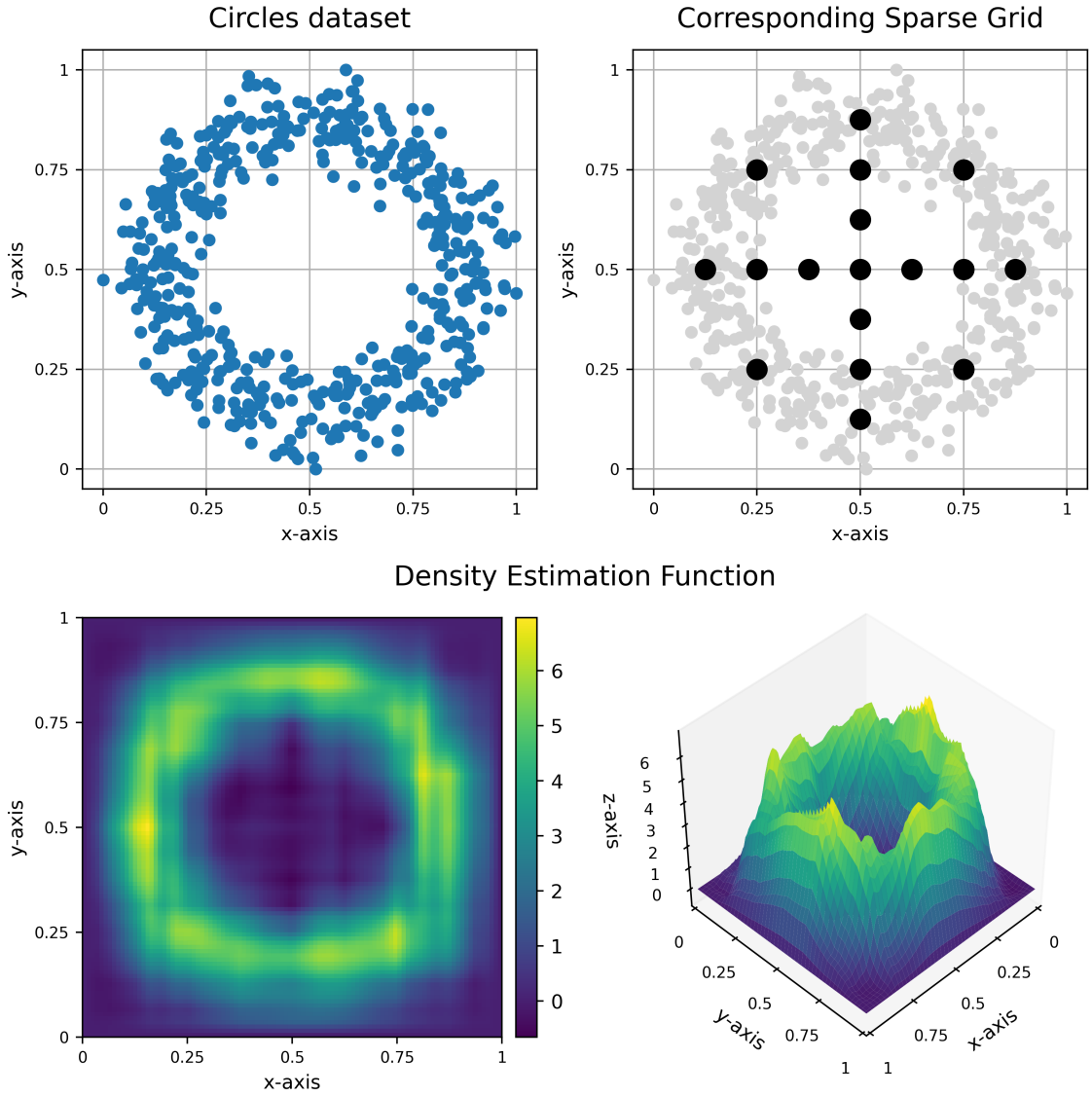


Figure 2.1: Construction of a density function on the "Circles" data set (taken from [6]), using a sparse grid of level 3. The data points (top left) are mapped onto a sparse grid (top right). The basis functions of the sparse grid points are then used to estimate the density function (bottom). Source: [7].

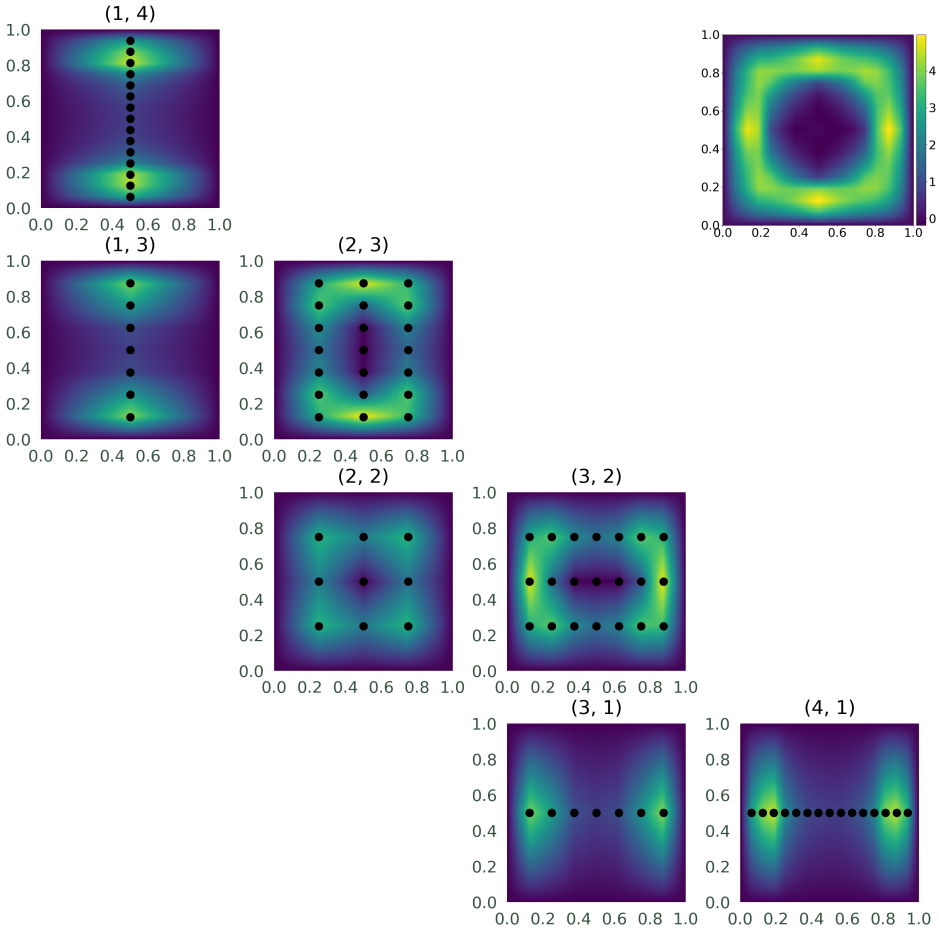


Figure 2.2: Construction of the density function on the "Circles" data set using the combination technique. The solution is computed on anisotropic full grids of different levels, ranging from $l_{min} = 1$ to $l_{max} = 4$, which are then combined to obtain the result (top right corner). Source: [7].

The first, overfitted estimate f_ϵ is constructed using

$$f_\epsilon = \frac{1}{M} \sum_{i=1}^{M} \delta_{x_i}, \tag{2.2}$$

with δ_{x_i} being the dirac delta function centered on x_i. Based on this initial guess, we then look for $\hat{f}$ in a suitable function space V:

$$\hat{f} = \underset{u \in V}{\operatorname{argmin}} \int_\Omega (u(x) - f_\epsilon(x))^2 dx + \lambda ||Lu||_{L^2}^2. \tag{2.3}$$

The left term ensures a closeness to f_ϵ, $||Lu||_{L^2}^2$ imposes a smoothness constraint and the *regularization parameter*$\lambda > 0$ balances smoothness and fidelity. This is then transformed into

$$\int_\Omega u(x) \cdot s(x)dx + \lambda \int_\Omega Lu(x) \cdot Ls(x)dx = \frac{1}{M} \sum_{i=1}^{M} s(x_i), \tag{2.4}$$

for all test functions $s \in V$. To solve this, the finite-dimensional function space $V_N \subset V$ is defined as the span of the basis functions $\Phi = \{\phi_1, \ldots, \phi_N\}$ centered at grid points. An approximated density function $\hat{f}_N \in V_N$ is then depicted as a linear combination of basis functions ϕ_i and coefficients α_i:

$$\hat{f}_N(x) = \sum_{i=1}^{N} \alpha_i \phi_i(x). \tag{2.5}$$

A sparse grid discretization is used to make this computationally feasible: Let Φ be the set of hierarchical basis functions of the sparse grid space $V_\ell^{(1)} \subset H_{mix}^2$ of level $\ell \in \mathbb{N}$. Incorporating this into (2.4) yields

$$\int_\Omega \hat{f}_N(x) \cdot \phi(x)dx + \lambda \int_\Omega L\hat{f}_N(x) \cdot L\phi(x)dx = \frac{1}{M} \sum_{i=1}^{M} \phi(x_i), \tag{2.6}$$

which can be simplified because $\hat{f}_N$ is a linear combination of basis functions Φ with coefficients α, yielding

$$(R + \lambda C)\alpha = b, \tag{2.7}$$

where R, C and b are computed using

$$R_{ij} = (\phi_i, \phi_j)_{L^2} = \int_\Omega \phi_i(x) \cdot \phi_j(x)dx, \tag{2.8}$$

$$C_{ij} = (L\phi_i, L\phi_j)_{L^2} = \int_\Omega L\phi_i(x) \cdot L\phi_j(x)dx, \tag{2.9}$$

$$b_i = \frac{1}{M} \sum_{j=1}^{M} \phi_i(x_j). \tag{2.10}$$

C may then be replaced by the identity matrix $\mathbf{I}$ for the purpose of penalizing non-smooth functions:

$$(R + \lambda \mathbf{I})\alpha = b. \tag{2.11}$$

In order to compute the density estimation function $\hat{f}_N$, this equation has to be defined for the desired sparse grid level $\ell \in \mathbb{N}$ and then solved for the coefficient vector α. With α and the respective basis functions Φ of the defined sparse grid space $V_\ell^{(1)}$ the density estimation function $\hat{f}_N$ is then constructed as in (2.5) [3].

2.2 Classification

Classification is a type of *supervised machine learning*. This means that the algorithm is given a *training set* $S = \{(x_i, y_i)\}_{i=1}^{N}$, consisting of N input-output pairs called *training examples*, and the goal is to learn a mapping from the inputs x_i to outputs y_i. The x_i and y_i can both theoretically be anything. When the y_i are categorical, the problem is called classification. In this case the outputs $y_i \in \{1, \dots, C\}$ are *classes*, also called *labels*, with C being the amount of classes. When $C = 2$, meaning that there are only two classes, it is called *binary classification*, while using more classes, with $C > 2$, is called *multiclass classification*. It may also be possible to map an input x to more than one class label y, this is then called *multi-label classification*. When not stated otherwise, "Classification" usually refers to multiclass classification with a single output. The goal is ultimately to assign classes to new inputs x, using a mapping learned from the training set S. Such a mapping can be learned using *function approximation*: The assumption is that there is some unknown function g that maps inputs to outputs, with $y = g(x)$. This function is estimated using S, yielding $\hat{g}$, which we can then use to assign approximated outputs $\hat{y}$ to new inputs x: $\hat{y} = \hat{g}(x)$ [1]. One approach to this is to use density estimation. This works as follow: First, the training set S is split into partitions, where each partition contains only input-output pairs of the same class k. There is one partition for each class, yielding C partitions $S^1, \dots, S^C$ with

$$S^k = \{(x_i, y_i) \in S | y_i = k\}. \tag{2.12}$$

Then, a probability density function $\hat{f}^k$ is estimated for each set S^k, as presented in Section 2.1. When assigning a new point x the estimated density functions are evaluated at x and the point is assigned to the class k where $\hat{f}^k(x)$ has the highest value:

$$y = \underset{k \in \{1,\dots,C\}}{\operatorname{argmax}} \hat{f}^k(x). \tag{2.13}$$

The confidence of this assignment depends on how much the evaluations of $\hat{f}^k(x)$ differ: If the highest value is significantly greater than the other values, x can be assigned to the corresponding class with great confidence. However if several density functions yield similar values, the mapping is less confident. This could, for example, occur for points x that lie in between classes [8]. Figure 2.3 depicts the "Moons" data set (taken from [6]), which is split into two partitions as explained in Equation 2.12. A density function is then estimated on both, as visualized in Figure 2.4.

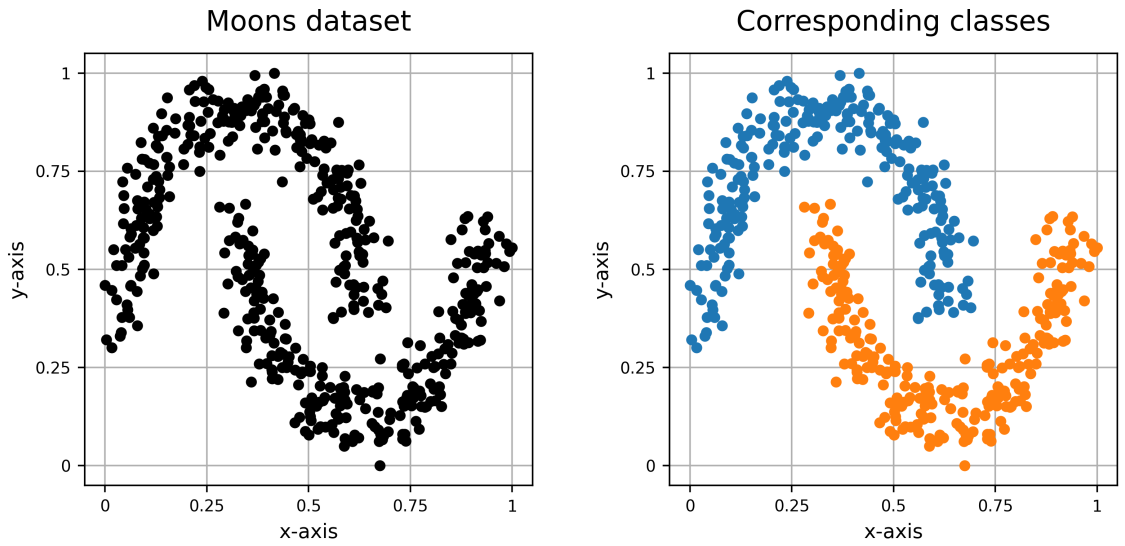


Figure 2.3: The "Moons" training set (left) is split into two partitions, depicted in blue and orange respectively (right). Source: [7].

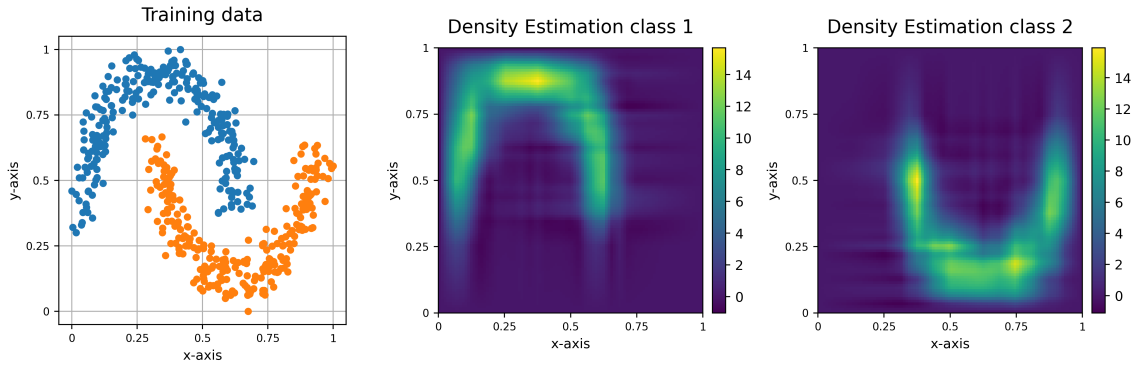


Figure 2.4: A density function is estimated on each partition, which can then be used to map new points x to classes as in Equation 2.13. Source: [7].

2.3 Clustering

Clustering is a type of *unsupervised learning*. The difference to supervised learning is that the given data $S = \{x_i\}_{i=1}^{N}$ contains only input values x, but no corresponding output values y. The goal is to find "interesting structure" in the given data. Unsupervised learning is generally more complicated than supervised learning, but it is more applicable since it does not require labels, which may be hard to obtain. Clustering is the task of creating groups, or *clusters* for input points with similar properties [1]. This can also be done with the help of density estimation: A region has high density when it contains many data points, and low density when it contains few. A cluster can be characterized as a region of high

density surrounded by a region of low density. To find such regions, the data is represented as a similarity graph $G = (S, E)$ with the given data points as vertices. Then a density function $\hat{f}$ is estimated for S. Lastly, all vertices x_i at which $\hat{f}(x_i) < \epsilon$ for some density threshold ϵ, are deleted, as well as their related edges. This way all vertices and edges inside of low-density regions are removed, splitting the graph into several connected components that represent the clusters. After obtaining the clusters in this manner, the points that were deleted in the process can now either be connected to the nearest cluster or considered noise and left out entirely [9].

3 Hyper-Parameter Optimization

This Chapter will introduce hyper-parameters and the problem of *hyper-parameter optimization*. It will then explain some of the most important methods to solve this problem, focused on *Bayesian Optimization*.

3.1 Introduction to Hyper-Parameter Optimization

Machine-Learning algorithms usually have parameters that need to be set beforehand and can not be updated during the learning phase. Such parameters are called hyper-parameters [2]. Examples would be the regularization parameter λ used for sparse grid density estimation in Equation 2.3 of Section 2.1, or the density threshold used for clustering in Section 2.3. These parameters can have a significant influence on training accuracy and speed, which is why finding good values for them is crucial. This can be done manually if the necessary expert knowledge about the respective hyper-parameters is available, which it is often not. The process of automatically optimizing hyper-parameters and largely removing humans from this task is called hyper-parameter optimization. Mathematically, hyper-parameter optimization seeks to optimize an objective function mapping the hyper-parameters to an evaluation, seeking to achieve maximum accuracy, or minimum loss. The result of the optimization process is then the set of hyper-parameters $x*$ that evaluates best on the objective function:

$$x* = \underset{x \in X}{\operatorname{argmax}} f_{obj}(x). \tag{3.1}$$

In order to evaluate this objective function for a specific setting of hyper-parameters, the algorithm has to be applied with these settings, training a model with a training data set and then evaluating it on a validation set. This makes evaluating the objective function, and therefore the whole process of hyper-parameter optimization expensive to compute [2]. This Chapter introduces several methods for hyper-parameter optimization and discusses their advantages and disadvantages.

3.1.1 Grid Search

Grid search performs an exhaustive search on a specified set of hyper-parameters, which I will refer to as the *search space*. This search space is generated by the users, who need to have some preliminary knowledge on the hyper-parameters in order to generate suitable candidates. This method is very expensive to compute for large search spaces, since all specified candidates are evaluated. It is therefore desirable to keep the search space narrow, which in effect means that the users need to have sufficient knowledge to define such a search space. Grid search is straightforward and mathematically simple. It also guarantees that the optimal combination of hyper-parameters is found, as long as this optimum is included in the search space, since all candidates will be tested. Another advantage is that it is easy

to be executed in parallel, since trials of different hyper-parameter combinations do not depend on each other. Drawbacks are that preliminary knowledge is needed, potentially to an extend that is not available, in order to define a search space that is sufficiently narrow but still includes the optimum. It also requires many evaluations of the objective function, making it expensive to compute [2].

3.1.2 Random Search

Random search randomly searches through possible values of hyper-parameters until either the desired accuracy is reached, or a given budget is depleted [2]. One advantage over grid search is that the search space does not have to be narrow, since the amount of values tried depends on the given budget and not on the size of the search space. In effect, one does not need as much knowledge about the hyper-parameters. For a limited amount of tries random search is statistically more likely to find the optimum, since it can usually not be guaranteed that the optimum is included in a manually generated search space. If it is not included, it can definitely not be found by grid search [10]. It can, however, not be guaranteed that the optimum is found using random search either. Similarly to grid search, random search is easy to parallelize since the evaluations do not depend on each other. It is still expensive since it requires a lot of evaluations of the objective function, but this is bounded by the given budget. Newly designed algorithms are often compared to random search, making it a baseline to measure the efficiency of hyper-parameter optimization methods [2].

3.1.3 Other Methods

There are numerous other methods for hyper-parameter optimization that may also be considered when selecting an algorithm for a specific case. This methods include, but are not limited to *population based training*, which is similar to *genetic algorithms* [11], and the use of an *early stopping policy* [2]. The choice of algorithm depends on the given resources and the specific problem: For example, some algorithms may only work with continuous values of hyper-parameters, but some problems use discrete values like integers or boolean.

3.2 Bayesian Optimization

Both grid search and random search require many evaluations of the objective function, making them expensive to compute. Bayesian Optimization seeks to reduce the amount of evaluations necessary, in order to decrease the computational cost. This section introduces the general concept of Bayesian optimization, and explains the specific type I have implemented, which will be discussed in more detail in Chapter 4.

3.2.1 Introduction to Bayesian Optimization

Bayesian Optimization is an optimization method for functions that can only be observed through input-output observations that may also be distorted by noise, also known as *noisy black box functions*. It is very data efficient and therefore especially useful when evaluating the function is expensive and not much information about it is given. This makes Bayesian optimization very helpful for the problem of hyper-parameter optimization: As

described in Section 3.1 the objective function is expensive to evaluate, and often the only information available on it is input-output behavior. This is also a significant advantage over grid search: Bayesian optimization does not require users to have knowledge on the hyper-parameters, as the algorithm obtains the needed information itself. The fundamental idea of Bayesian optimization is to learn from the information obtained by prior evaluations, in order to make suitable assumptions about where the optimum may be [12]. For example, if all prior evaluations in a specific area were significantly better than evaluations in other areas, it does intuitively make sense to expect the optimum to be in the vicinity of this area. This is the underlying principle of Bayesian optimization [13]. Each iteration of the algorithm looks as follow: The prior input values of the function $x_1, \ldots, x_n$, as well as their respective evaluations $y_1, \ldots, y_n$ make up the *evidence set* D_n. They are used to generate a statistical model of the objective function, called the *surrogate model*. Based on this model, an *acquisition function* $\alpha_n(x)$ is created. Maximizing this acquisition function yields the next value that should be evaluated on the objective function f_{obj}:

$$x_{n+1} = \underset{x \in X}{\operatorname{argmax}}\ \alpha(x; D_n). \tag{3.2}$$

After evaluating x_{n+1} on the objective function, it and the respective $y_{n+1} = f_{obj}(x_{n+1})$ are both added to the evidence set. The statistical model is updated and then used to create the new acquisition function α_{n+1}, which is then again used to obtain the next value to be evaluated on the objective function. These steps are repeated until a given budget is depleted, or the optimum is found. Bayesian optimization is mathematically less straightforward than grid search and random search, but the additional computational effort is compensated by the reduced amount of evaluations of the objective function needed.

Algorithm 1: Bayesian optimization

```
for n = 1, 2, ... do
    // select new x_{n+1} by optimizing acquizition function α
    x_{n+1} = argmax_{x∈X} α(x; D_n)
    // query objective function to obtain y_{n+1}
    y_{n+1} = f_obj(x_{n+1})
    // augment evidence set
    D_{n+1} = {D_n, (x_{n+1}, y_{n+1})}
    UPDATE_STATISTICAL_MODEL()
```

Figure 3.1: Pseudocode of Bayesian optimization.

The key ingredients of Bayesian optimization are the surrogate model and the acquisition function. An in-depth discussion about different choices for both can be found in [12]. Popular options for the surrogate model include *random forests*, *tree parzen estimators* and *Gaussian process*. Which model is best depends on the properties of the specific search space, for instance what data types the hyper-parameters are and how much they depend on each other. The acquisition functions differ in their exact methods, but most of them balance *exploration* and *exploitation*. Exploration is the process of gaining new information to best improve the surrogate model, by evaluating points where the model is uncertain.

Exploitation uses the current model to find the point where the optimum is most likely to be. Among others, the types of acquisition functions include *improvement based policies* and *optimistic policies*. The former favors points that are likely to lead to an improvement of the current optimum. This type of method includes *probability of improvement* and *expected improvement*, where either the likelihood of improvement or the expected amount of improvement is maximized. *Optimistic policies* use the best-case scenario according to the current model, even though this scenario is uncertain - hence the name "optimistic". This type includes *upper confidence bound* and *Gaussian process upper confidence bound* (GP-UCB) [12].

3.2.2 Discrete Bayesian Optimization using GP-UCB

In my implementation I have used a Gaussian process to create the surrogate model and GP-UCB as the acquisition function. This Gaussian process $GP(\mu(x), k(x, x'))$ is defined by two ingredients: its *mean function* $\mu(x)$, which may be assumed to be a zero function in the context of Bayesian optimization, and its *covariance function* $k(x, x')$. In the context of this thesis the used covariance function is the *squared exponential* kernel

$$k(x, x') = \sigma_k^2 exp(-\frac{1}{2l_n^2}||x - x'||^2), \tag{3.3}$$

where σ_k^2 dictates the uncertainty in the objective function, and the length scale parameter l_n controls how quickly a function can change [14]. Both can either be optimized in each iteration [15], or set to constants for simpler computation but a loss of efficiency. A variety of other common choices for kernel functions is given in Section 5.2. of [16]. Fitting the evidence set $D_n = \{(x_i, y_i)\}_{i=1}^N$ into the Gaussian process yields the *predictive distribution* of the objective function f_{obj}, which is also a Gaussian process $GP(\mu(x), \sigma^2(x))$ with the mean

$$\mu(x) = \mathbf{k}^T (K + \sigma_\epsilon \mathbf{I})^{-1} \mathbf{y}, \tag{3.4}$$

and the variance

$$\sigma^2(x) = k(x, x) - \mathbf{k}^T (K + \sigma_\epsilon \mathbf{I})^{-1} \mathbf{y}. \tag{3.5}$$

In this context, $\mathbf{k} = [k(x_i, x)]_{\forall x_i \in D_n}$ is a vector of the covariances of the new point x and all other points x_i included in the current evidence set, $K = [k(x_i, x_j)_{\forall x_i, x_j \in D_n}]$ is the covariance matrix, σ_ϵ is a parameter denoting measurement noise, $\mathbf{I}$ is the identity matrix of the same dimension as K, and $\mathbf{y} = (y_i, \ldots, y_n)$ is a vector consisting of all evaluations of the objective function obtained so far. The acquisition function using GP-UCB is then

$$\alpha_n(x) = \mu(x) + \sqrt{\beta_n}\sigma(x). \tag{3.6}$$

In simple terms, $\mu(x)$ can be said to represent exploitation, $\sigma(x)$ exploration, and β_t is the *exploitation-exploration trade-off factor*. The specific calculation of the β_n depends on the context. If the search space is a subset of $[0, r]^d$, with dimension $d \in \mathbb{N}$ and radius $r > 0$, β_n is proposed to be

$$\beta_n = 2log(\frac{n^2 2\pi^2}{3\delta}) + 2d\ log(n^2 dbr\sqrt{log(\frac{4da}{\delta})}) \tag{3.7}$$

for $\delta \in (0, 1)$ and constants $a, b > 0$, the choice of which again depends on the exact context [16]. It is also crucial to mention that the iterations need to begin with $n = 1$, as β_n is undefined for $n = 0$. The next point is then acquired by maximizing $\alpha_n(x)$:

$$x_{n+1} = \underset{x \in X}{\operatorname{argmax}}\ \alpha_n(x). \tag{3.8}$$

An issue with this acquisition function is that it assumes that the objective function is continuous. This is, however, not always the case: If the hyper-parameters assume discrete values like integer values or boolean, the search space and the objective function are also discrete. The assumption that the objective function is continuous is a common problem with acquisition functions, as for example probability of improvement and expected improvement suffer from this as well. The first approach to solving this is to use *naive rounding*. This simply rounds the value acquired by Equation 3.8 to the nearest value included in the discrete search space. This approach may result in repeatedly evaluating the same point, as rounding a new point $x_{n+1} \notin C_n$ may give a point that is already in the evidence set $rd(x_{n+1}) \in C_n$. When this happens, the algorithm is unable to acquire new information and gets stuck at this point. That is why the authors of [14] propose another method: Fundamentally, increasing β_n, and therefore the factor of exploration in the acquisition function, should always yield a new x. In order to not have to use excessively high values for β_n, the length-scale parameter l_n used in the covariance function can also be adjusted. When the acquisition function yields values that are, after rounding, already in the evidence set, this method finds new values β_n^*, l_n^* by minimizing a function $g(\beta_n + \Delta\beta, l)$ that models the desired objectives for the new values:

$$\beta_n^*, l_n^* = \underset{\Delta\beta \in [0, \beta_h], l \in [0, l_h]}{\operatorname{argmin}} g(\beta_n + \Delta\beta, l), \tag{3.9a}$$

$$g(\beta_n + \Delta\beta, l) = \Delta\beta + ||x_{n+1} + x'_{n+1}||_2 + P(x'_{n+1}). \tag{3.9b}$$

Here, x_{n+1} is the point acquired by using α_n with the original β_n and l_n, where $rd(x_{n+1}) \in D_n$. The x'_{n+1} is the value that α_n suggests using the updated $\beta_n + \Delta\beta$ and l. There are also upper bounds β_h and l_h that can be set manually. The objectives modeled in $g(\beta_n + \Delta\beta, l)$ are as follows: $\Delta\beta$ should be small, so that the new β_n^* does not become too big. Secondly, the new x'_{n+1} should not be too far away from the original x_{n+1}, since that is where the acquisition function originally assumed the current optimal value for evaluation to be. Lastly, we do not want the new x value acquired using β_n^* and l_n^* to also already be in the evidence set. This is modeled by the penalty function $P(x'_{n+1})$: This function is zero if $rd(x'_{n+1}) \notin C_n$, and a constant otherwise. Using this approach, the algorithm should find a new value for x and not get stuck at one point [14].

4 Implementation

This Chapter introduces the sparseSpACE framework [17], with a focus on the implementations of the density estimation and machine learning algorithms. It will then continue to give a more detailed explanation of the implementation of hyper-parameter optimization methods added in the context of this thesis, that aim to optimize the hyper-parameters of the previously implemented machine learning algorithms.

4.1 Density Estimation and Machine Learning with the sparseSpACE-Framework

Sparse Grid Spatially Adaptive Combination Environment, short sparseSpACE, is a Python framework created by Michael Obersteiner. This framework includes various spatially adaptive combination techniques, as well as arbitrary grid based operations. In 2020, the `DensityEstimation` functionality, which implements the estimation of a density function (see Section 2.1), has been added by Lukas Schulte [18]. Based on that, the `DEMachineLearning` wrapper has been implemented by Cora Moser, which includes classification and clustering using density estimation, as explained in Section 2.2 and Section 2.3 [7]. In the same year, density estimation using the *dimension-wise spatially adaptive refinement* method as well as classification using single-dimensional refinement have been added by Markus Fabry [19].

To use the density estimation functionality, users create a `DensityEstimation` object. At this point, they need to specify the data the estimation should be performed on, which may, for example, be created using the `scikit-learn` [6] package `sklearn.datasets` that provides various types of datasets. In addition to the data, users can specify the regularization parameter λ to control the smoothness of the estimated density function. They can also turn *mass lumping* on or off: When this is turned on, the calculation of R (Equation 2.8) ignores the cases where the basis functions overlap only partially. This way, R is a diagonal matrix, simplifying the computation of Equation 2.11, but also decreasing the accuracy [18]. In order to use the classification functionality, users need to initialize a `Classification` object, on which they can then call the `perform_classification` method to start the learning process. They can also call the `perform_classification_dimension_wise` method if the classification process should be based on density estimation using the dimension-wise spatially adaptive refinement method. The `Classification` class requires the data it is supposed to perform classification on in the form of a `DataSet` object: as part of the `DEMachineLearning` wrapper, the `DataSet` class has been implemented to simplify working with the data. It includes functions such as separating the data based on their labels or scaling it. There are also optional parameters the user can specify when creating a `Classification` object, most notably the *splitting percentage* which determines how much of the data set is used for training purposes and how much for evaluation purposes. The training set is further

split into partitions with data of the same class to prepare for the classification learning process, as depicted in Equation 2.12. The `perform_classification` method requires λ, as well as a boolean parameter to determine the use of mass lumping, in order to create a `DensityEstimation` object to estimate density functions on all the partitions. Other parameters, such as the minimal and maximal level of the underlying sparse grid (see Figure 2.2), may be specified if desired but are optional. The same parameters are required for the `perform_classification_dimension_wise` method, but there are more optional parameters that can be used. Classification can be performed once on each `Classification` object, and later evaluated using the `evaluate` method. This method maps the testing data to classes using the estimated density functions, and then compares the calculated classes to the actual classes of this data to determine and finally return the amount of wrong mappings, the amount of total mappings and the percentage of correct mappings [7].

4.2 The HP_Optimization module

As an addition to the sparseSpACE framework I have implemented the `HP_Optimization` module. This module consists of the `HP_Optimization` class which contains methods for hyper-parameter optimization, as well as the `Optimize_Classification` class to support optimizing the hyper-parameters for the classification methods introduced in Section 4.1.

4.2.1 The HP_Optimization class

The `HP_Optimization` class has two required inputs: The objective function that needs to be optimized, as well as a hyper-parameter space that specifies the possible values of the hyper-parameters. The objective function can theoretically be any function, but in the context of hyper-parameter optimization it should be one that takes in a set of hyper-parameters, performs a machine learning task with them and returns an evaluation, as explained in Section 3.1. In order to optimize the hyper-parameters of a specific machine learning task, such an objective function needs to be created for it. The input of the objective function is called *x-value*, and the corresponding evaluation *y-value*. The hyper-parameter space is assumed to contain information about as many hyper-parameters as the objective function takes in, which is therefore also the dimension of the x-values. This information is encoded as a multi-dimensional array called `hp_space`, that contains one array for each hyper-parameter. These start with a string of the type of the specific hyper-parameter that specifies whether it can assume values within a continuous *interval*, integer values in an *interval_int*, or exactly the values given in the form of a *list*. This descriptor is then followed by values that either denote the start and end of the interval, or the values in the list. If only one value is given, this value is assumed to be the only one possible for this hyper-parameter, regardless of the type. This is handled similarly if the given type is unknown: In this case the first value given after the descriptor is assumed to be the only one possible. If a value is given instead of a descriptor, it is handled like an unknown type. If less information than one descriptor and one value is given it defaults to ['list', 1], which may however lead to issues during hyper-parameter optimization if the objective function does not work with this value. Similar problems may occur of not enough hyper-parameters are specified in `hp_space`. If the best possible evaluation of the objective function is known, for example 100% or 1 in the case of evaluating a classification process by the percentage of correct mappings, this may also be

Algorithm 2: Grid search

```
Function perform_GO(f_obj, search_space, interval_levels):
    if search_space not given then
        search_space = create_search_space(interval_levels)
    for x in search_space do
        y = f_obj(x)
        if y > best_y then
            best_y = y
            best_x = x
    return best_x, best_y
```

given to the `HP_Optimization` class through the parameter `f_max`. This is then used to stop the optimization process if this value is achieved. After creating a `HP_Optimization` object users can call one of three optimization methods: `perform_GO` for grid search, `perform_RO` for random search and `perform_BO` for Bayesian optimization. All three take in some kind of information about the x-values they are supposed to try, and return the best combination of hyper-parameters found called `best_x`, the evaluation of the objective function there called `best_evaluation`, as well as a list of all combinations tried and another list of all their evaluations to help retrace the process.

When calling `perform_GO`, users can either directly specify the search space as an array containing all x-values they want the algorithm to try, or they can give a value called `interval_levels`. If no search space is given, the method automatically creates a search space using this parameter and the information given in `hp_space` by taking roughly `interval_level+1` many values from intervals in `hp_space`, all values given in the form of lists and combining them using the Cartesian product. This, however, may result in a very large search space, and does not guarantee that the optimum is included in it. The method then proceeds to iterate through all values in the search space, evaluating all of them on the given objective function and returning x-value and evaluation of the one that evaluated best. Algorithm 2 provides a rough depiction of the `perform_GO` method in the form of pseudo-code.

When using `perform_RO`, the users simply need to state how many iterations should be performed through the parameter `amt_it`. In each iteration, the method creates a random x-value and evaluates the objective function on it. The random values are based on `hp_space`, and it is made sure that only values that follow the criteria specified there are created. After doing this `amt_it` many times, the best x-value found this way and its corresponding evaluation are returned. A brief depiction of the `perform_RO` method using pseudo-code is given in Algorithm 3.

Lastly, `perform_BO` also requires only the amount of iterations desired, `amt_it`. The optional parameters r, δ, a and b are used to calculate β (see Equation 3.7) and default to $r = 3$, $\delta = 0.1$, $a = 1$ and $b = 1$. The algorithm first initializes Bayesian optimization with $d + 1$ random points, with d being the dimension of the x-values, as proposed in [14].

Algorithm 3: Random search

```
1 Function perform_RO(f_obj, amt_it):
2     for n in range (0, amt_it) do
3         x = get_random_x()
4         y = f_obj(x)
5         if y > best_y then
6             best_y = y
7             best_x = x
8     return best_x, best_y
```

These random points, along with their evaluations, make up the initial evidence set. In each iteration the current evidence set is used to calculate the surrogate model based on a Gaussian Process using the squared exponential kernel, as well as the acquisition function $\alpha(x)$ using GP-UCB, as explained in Subsection 3.2.2. The parameters of the kernel are chosen to be constants $\sigma_k = 1$ and $l_n = 0.5$, and the measurement noise is set to $\sigma_\epsilon = 0$. This acquisition function is then maximized to obtain `new_x`, as well as rounded to get `new_x_rd`, a value that meets the criteria specified in `hp_space`. If `new_x_rd` is not yet included in the evidence set, it is evaluated on the objective function and added along with its evaluation. Then the new iteration is started, again calculating the surrogate model and the acquisition function, but based on the updated evidence set.

If the current evidence set already includes `new_x_rd`, new values for β and l are obtained as explained by Equation 3.9. This does, however, not always help to find a new x-value that is not already in the evidence set. For example, this may occur if all possible values specified in `hp_space` are already included in the evidence set. Unfortunately, it is often unclear why no new value can be found and may be due to numerical inaccuracies or a bug in the code. If a new x-value is found this way, it and its evaluation are added to the evidence set. If it is not, the same process to get a new β and l is tried again twice, each time with increased β_h als l_h so that more values are tried, as well as an increased penalty for points x that are already in the evidence set. After that, if still no new x-value has been found, the algorithm instead generates a random x. If even this way no value can be found that is not already in the evidence set, the algorithm assumes that `hp_space` has been exhausted and terminates. Otherwise, the random x and its evaluation are added to the evidence set. This way, some information is gained that may help to find new points in the remaining iterations of Bayesian optimization. To help with retracing the steps of the algorithm, it records in which steps a random value was used in the form of an array called `used_random_x`, and writes this information to the console at the end. After adding the random x-value and its evaluation to the evidence set, the algorithm continues to the next iteration. After finishing all iterations, `perform_BO` returns the x-value where the best evaluation has been found, the corresponding evaluation, the x- and y-values of the evidence set as well as `used_random_x`. A conceptual pseudo code of `perform_BO` can be found in Algorithm 4.

Algorithm 4: Bayesian optimization

```
Function perform_BO(f_obj, amt_it, r, δ, a, b):
    // Create initial evidence set C with random values
    C = create_initial_evidence_set()
    for n in range (0, amt_it) do
        // get_l always returns a specified constant
        l = get_l(n)
        β = get_beta(n, r, δ, a, b)
        x = acquire_new_x(C, n, β, l)
        // If the x is already in C, get new β and l to get new x
        if x ∈ C then
            new_β, new_l = get_new_beta_and_l()
            x = acquire_new_x(C, n, new_β, new_l)
        // If new x is also in C, get random x
        if x ∈ C then
            x = get_random_x()
        y = f_obj(x)
        if y > best_y then
            best_y = y
            best_x = x
        // Add new values to the evidence set
        C.add(x, y)
    return best_x, best_y
```

4.2.2 The Optimize_Classification class

The `Optimize_Classification` class was created to support hyper-parameter optimization of the classification functionality introduced in Section 4.1. It provides objective functions for classification and dimension-wise classification, as well as hyper-parameter spaces that can be used as `hp_space` for both. Creating an `Optimize_Classification` object requires no parameters, but if desired, users may specify the data on which classification should be performed. They can do so either in the form of a `DataSet` object or by specifying information so that a new data set can be created, such as the name of the type of data and the dimension. Creating a new data set by name offers only a small variety of options, so if a specific data set is desired it should be passed to the constructor of `Optimize_Classification` directly. The data set defaults to the "Moons" data set. Users can also optionally specify some parameters that are used for classification, but that should not be optimized, such as the maximal level of the underlying sparse grid in the density estimation process `max_lv` (see Figure 2.2), which is 3 per default. Each `Optimize_Classification` object has two functions that perform an evaluation at a specific set of hyper-parameters, and therefore take the role of objective functions. Both take in a list of parameters, create a new `Classification` object using the specified data, perform classification with the given hyper-parameters, and return the evaluation of this classification process in the form of

the percentage of correct mappings. The set of hyper-parameters given to the objective function are the parameters used for the specific type of classification, for example λ and mass lumping for both. The function `pea_classification` performs normal classification, while `pea_classification_dimension_wise` uses dimension-wise classification, with "pea" being short for "perform evaluation at". An `Optimize_Classification` object also provides two arrays called `classification_space` and `class_dim_wise_space`. When optimizing the objective functions, these arrays may be used as `hp_space`, with `classification_space` being meant for normal classification, while `class_dim_wise_space` is supposed to be used for dimension-wise classification. They are, however, not automatically used when their respective objective function is being optimized, so users can also specify different spaces if desired. When users want to optimize them, these functions, together with their respective spaces, can simply be used to create a new `HP_Optimization` object on which optimization can then be executed.

5 Results

In this Chapter, the performance of the hyper-parameter optimization methods presented in Chapter 4 will be analyzed and compared to that of the open source software *hyperopt* [20]. The hyper-parameters of `pea_classification` and `pea_classification_dimension_wise` on different data sets have been optimized using `perform_GO`, `perform_RO`, `perform_BO` and hyperopt. The results of all four will be presented and compared, in terms of how good the hyper-parameter input found is and how quickly it has been found. The data sets used for this are the "Circles" data set, the "Moons" data set, as well as the "Classification" data set, the "Blobs" data set and the "Gaussian Quantiles" data set of dimension 2 and 4. In [7], a short introduction to these data sets is given, as well as an assessment of their usage for classification purposes. When creating the data sets, their random state is set to 1, making them the same every time. This helps to compare the optimization algorithms, as they are used on the exact same problems.

5.1 Hyperopt

"Hyperopt is a Python library for serial and parallel optimization over awkward search spaces, which may include real-valued, discrete, and conditional dimensions" [20]. It provides random search, as well as hyper-parameter optimization based on tree parzen estimators [21]. Optimization is executed using the function `fmin`, which takes in a loss function, a search space, the preferred optimization algorithm and the maximal amount of evaluations that should be done in the process. The output of `fmin` is the input x for which the loss function is minimal [20]. This is different to my implementation, which searches for the maximum of an evaluation function. To adjust to this, hyperopt is not directly given one of the evaluation functions `pea_classification` and `pea_classification_dimension_wise`, but a function that returns the result of these functions subtracted from 1. Since the evaluation functions return the percentage of correct mappings, subtracting them from one gives the percentage of incorrect mappings, which should be minimized and therefore works as a loss function. After getting the input x that minimizes this loss function using `fmin`, its evaluation can be subtracted from 1 again in order to obtain a result that is comparable to the results of my implementation. Since both classification and dimension-wise classification have awkward hyper-parameter search spaces, with for example λ being real-valued and *mass lumping* being boolean and therefore discrete, hyperopt is a good choice for optimizing them.

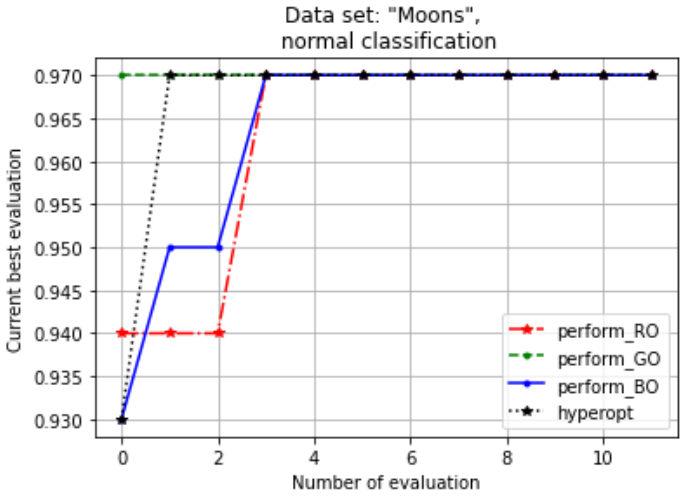
5.2 Optimization of Normal Classification

In order to compare the hyper-parameter optimization methods on classification using the specified data sets, an `Optimize_Classification` object is created for each data set. The `pea_classification` method and the `classification_space` of this object are then used to create a new `HP_Optimization` object, on which `perform_GO`, `perform_RO` and `perform_BO` are called. The search space for `perform_GO` is generated automatically using `interval_levels` = 2. This corresponds to a total of 12 evaluations in the case of normal classification, ranging from evaluation number 0 to number 11. The other methods are set to the same total amount of evaluations. Additionally, hyperopt is set up with a loss function that returns $1 -$ `pea_classification`, a search space that corresponds to `classification_space` and 12 as the maximal amount of evaluations. Figure 5.1 describes the hyper-parameters of normal classification as well as the values they can have, as encoded in `classification_space`. It should be noted that the minimal level of the underlying sparse grid, `min_lv` is technically not optimized, as only one possible value for it is specified. This is because using a higher value for it significantly increases the computational cost of classification and therefore hyper-parameter optimization. It could, however, be optimized as well if users define a different search space with more options for `min_lv`. The maximal level is specified at the creation of the `Optimize_Classification` object and set to `max_lv` = 5 for normal classification. Also noteworthy is that the input value for `lambd_exp` is not directly used for the regularization parameter λ, but $\lambda = 1^{-(\texttt{lambd\_exp})}$, making λ smaller when `lambd_exp` is bigger, and $\lambda \in (0, 1]$.

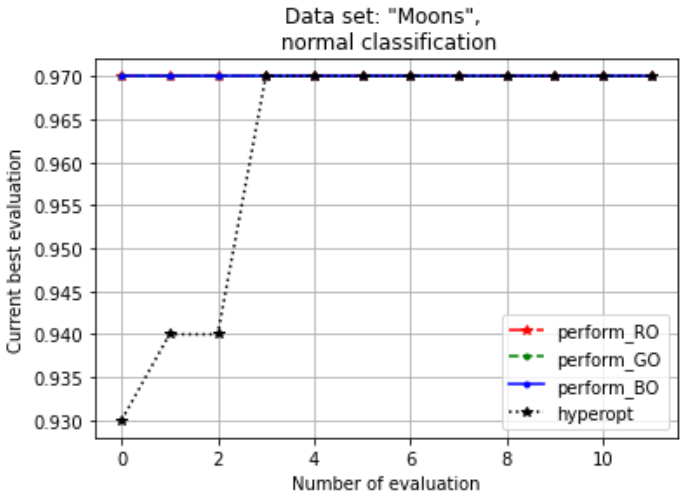
Name	Type	Values
`lambd_exp`	float	[0, 20]
`massl`	bool	{0, 1}
`min_lv`	int	{1}
`one_vs_others`	bool	{0, 1}

Figure 5.1: The hyper-parameters of normal classification.

When comparing the four methods, it should be kept in mind that `perform_RO` and `perform_BO` both use random x-variables, making their performance different every time they are used. This can be seen when using them on the same data set twice, as depicted by Figure 5.2. Normal classification on the "Moons" data set is used both times, but `perform_RO` and `perform_BO` both perform significantly better in Figure 5.2b than they do in Figure 5.2a, as they reach the evaluation of 0.97 faster. In both cases, `perform_GO` performs the same, as it always searches through the same values in the same order. Hyperopt can also be seen to have an element of randomness, as it reaches an evaluation of 0.97 faster in Figure 5.2a than it does in Figure 5.2b. In both test runs, all optimization algorithms reach the evaluation 0.97 in the given amount of iterations. In Figure 5.2a it is even reached within the first 4 iterations, which is fairly fast.



(a) Example of random search and Bayesian optimization reaching an evaluation of 0.97 within four evaluations.



(b) Example of random search and Bayesian optimization reaching an evaluation of 0.97 within one evaluation.

Figure 5.2: Comparison of optimization algorithms using normal classification on the "Moons" data set twice. The x-axis depicts the number of the current evaluation starting with number 0, while the y-axis depicts the best evaluation found so far.

Another interesting aspect to look at is how often `perform_BO` had to use random values. As explained in Subsection 4.2.1, when the acquisition function gets stuck at one point, the last resort to get a new value is to generate a random x-value. At the end of `perform_BO` it is written to the console and returned in what steps random values have been needed, but it is not depicted in the graphs. In the case of normal classification on the "Moons" data set, random values have been used in every step. In such cases `perform_BO` is unfavorable in comparison to `perform_RO`, as both only use random values, but `perform_BO` requires more computational effort.

It is also interesting to compare different input points and their evaluations, as this may give an insight as to which hyper-parameters are more important than others, and whether specific settings seem to generally be good. For this purpose, Figure 5.3 gives all the best values for normal classification on the "Moons" data set that have been found by the different optimization algorithms, and Figure 5.4 all the values evaluated by `perform_GO` together with their evaluations. Both are taken from the same run as the values for Figure 5.2. In this case, it is noteworthy that two of four best input values found have `massl` $= 1$ and `one_vs_others` $= 0$, raising the suspicion that these values may be particularly good. However, the evaluations in Figure 5.4 show that there are several other values for which the evaluation of 0.97 is reached. The values there suggest that `lambd_exp` $= 0.0$ performs particularly well, as all such input values have an evaluation of 0.97. Additionally, all other values that reach an evaluation of 0.97 have `massl` $= 1$, suggesting that this value may also be particularly good.

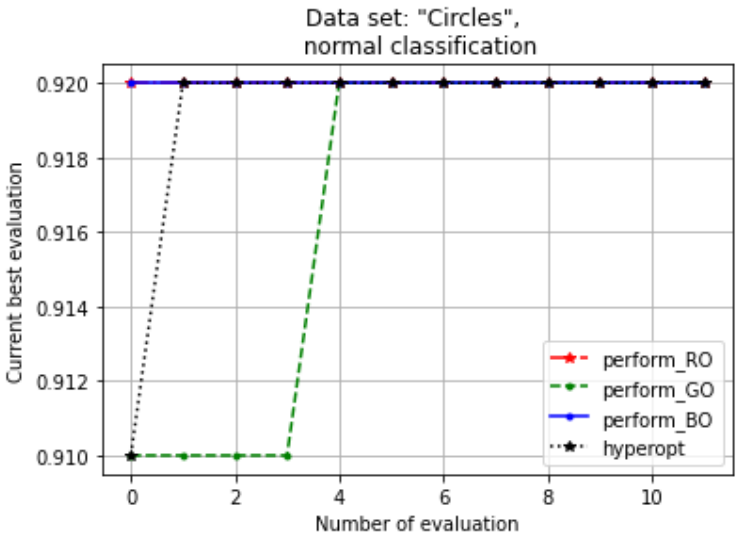
Algorithm	lambd_exp	massl	min_lv	one_vs_others
GO	0.0	0	1	0
RO	2.52..	1	1	0
BO	14.51..	1	1	0
hpo	6.92..	1	1	1

Figure 5.3: The best input values found for normal classification on the "Moons" data set.

No.	lambd_exp	massl	min_lv	one_vs_others	evaluation
0	0.0	0	1	0	0.97
1	0.0	0	1	1	0.97
2	0.0	1	1	0	0.97
3	0.0	1	1	1	0.97
4	10.0	0	1	0	0.94
5	10.0	0	1	1	0.929..
6	10.0	1	1	0	0.97
7	10.0	1	1	1	0.97
8	20.0	0	1	0	0.94
9	20.0	0	1	1	0.929..
10	20.0	1	1	0	0.97
11	20.0	1	1	1	0.97

Figure 5.4: All input values evaluated by `perform_GO` for normal classification on the "Moons" data set, in the order they were evaluated, together with their evaluations.

In Figs. 5.5 to 5.8 the optimization algorithms are compared on normal classification on the other data sets. For each data set, a graph like Figure 5.2 is given, which depicts the best evaluation found in a specific number of evaluations by each optimization algorithm. The input values where this best evaluation was reached is then given in a table similarly to Figure 5.3, but with shortened notations. In these tables, "`GO`", "`RO`", "`BO`" and "hpo" stand for `perform_GO`, `perform_RO`, `perform_BO` and hyperopt, while "Algo." is short for "Algorithm", "λ_exp" is `lambd_exp`, "`mnlv`" is `min_lv` and "`ovo`" stands for `one_vs_others`.

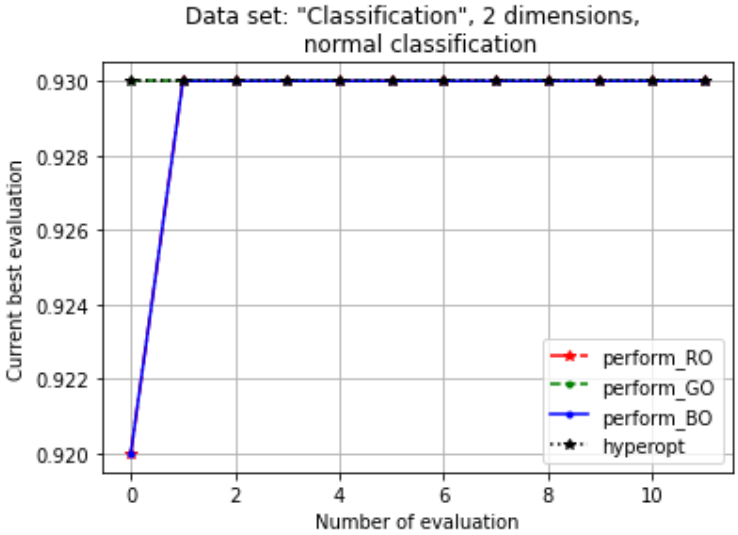


(a) Normal classification on the "Circles" data set.

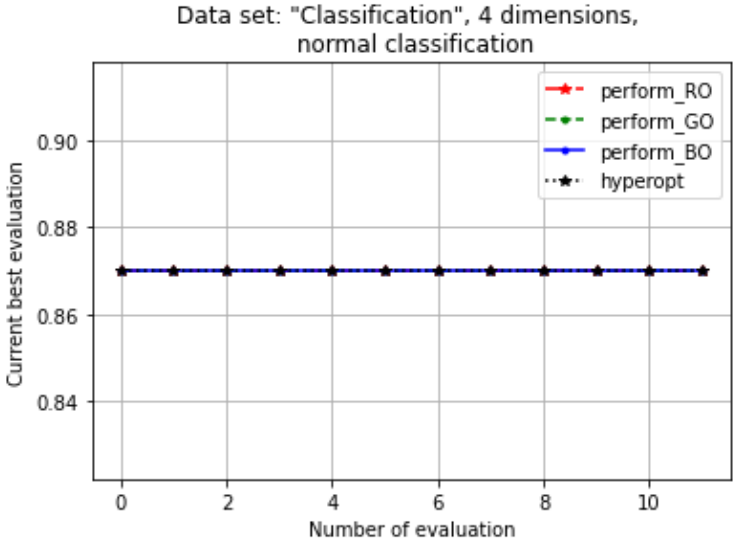
Algo.	λ_exp	massl	mnlv	ovo
GO	10.0	0	1	0
RO	16.57..	0	1	0
BO	9.21..	0	1	0
hpo	8.21..	0	1	0

(b) The best input values found for normal classification on the "Circles" data set.

Figure 5.5: Comparison of optimization algorithms using normal classification on the data sets "Circles"



(a) Normal classification on the "Classification" data set of dimension 2.



(b) Normal classification on the "Classification" data set of dimension 4.

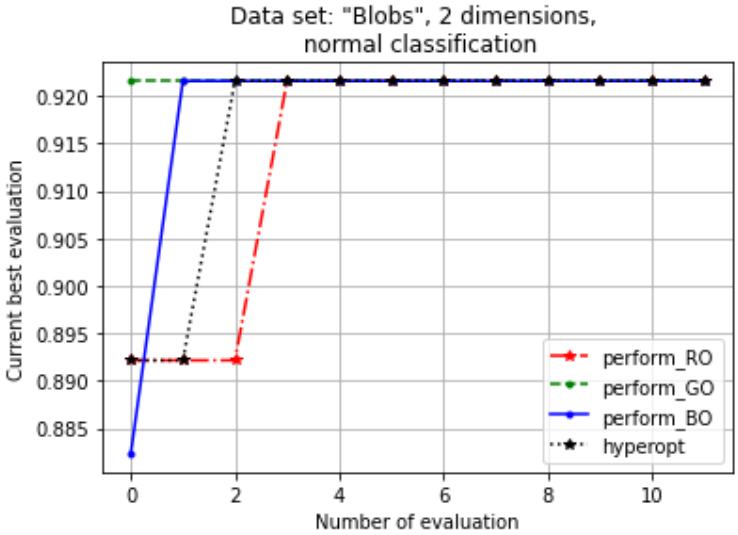
Algo.	λ_exp	massl	mnlv	ovo
GO	0.0	0	1	0
RO	0.35..	0	1	1
BO	1.76..	1	1	1
hpo	1.4..	0	1	0

(c) The best input values found for normal classification on the "Classification" data set of dimension 2.

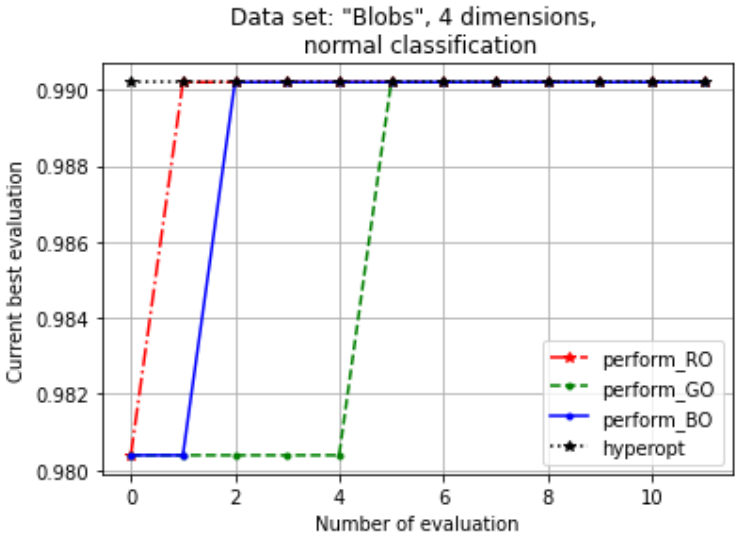
Algo.	λ_exp	massl	mnlv	ovo
GO	0.0	0	1	0
RO	17.32..	1	1	0
BO	4.79..	1	1	1
hpo	10.03..	1	1	0

(d) The best input values found for normal classification on the "Classification" data set of dimension 4.

Figure 5.6: Comparison of optimization algorithms using normal classification on the "Classification" data sets of dimensions 2 and 4.



(a) Normal classification on the "Blobs" data set of dimension 2.



(b) Normal classification on the "Blobs" data set of dimension 4.

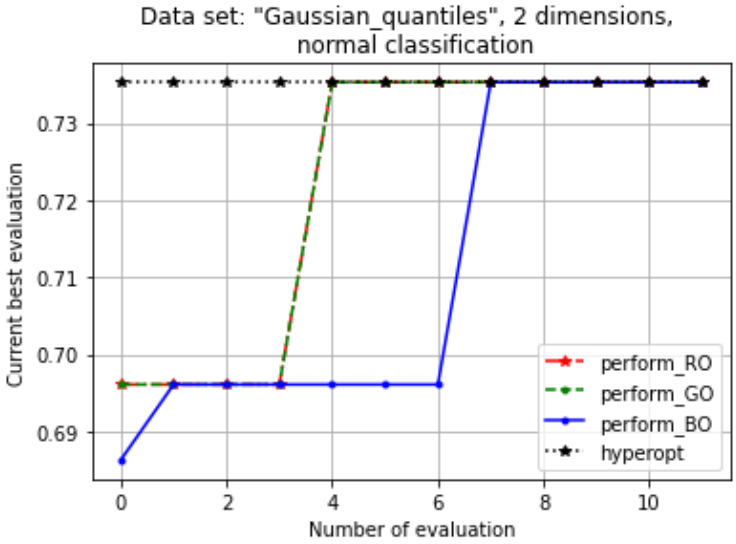
Algo.	λ_exp	massl	mnlv	ovo
GO	0.0	0	1	0
RO	15.84..	1	1	0
BO	17.01..	1	1	0
hpo	3.11..	1	1	0

(c) The best input values found for normal classification on the "Blobs" data set of dimension 2.

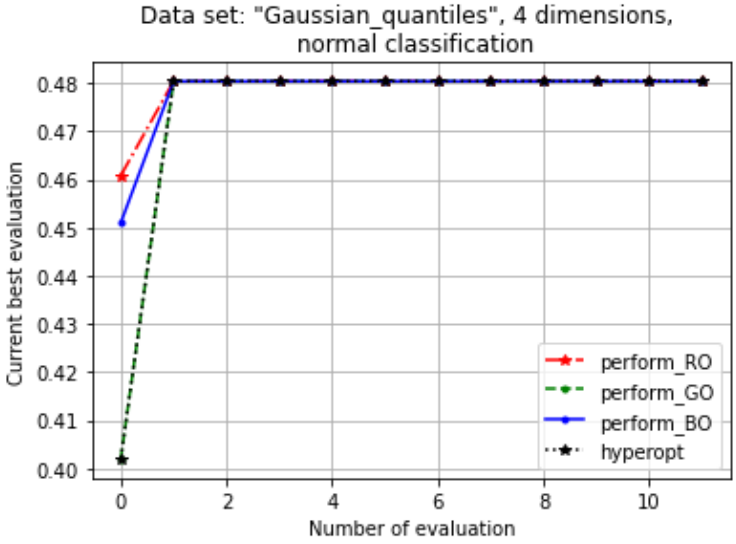
Algo.	λ_exp	massl	mnlv	ovo
GO	10.0	0	1	1
RO	7.6..	0	1	1
BO	19.91..	0	1	1
hpo	19.05..	0	1	1

(d) The best input values found for normal classification on the "Blobs" data set of dimension 4.

Figure 5.7: Comparison of optimization algorithms using normal classification on the "Blobs" data sets of dimension 2 and 4.



(a) Normal classification on the "Gaussian Quantiles" data set of dimension 2.



(b) Normal classification on the "Gaussian Quantiles" data set of dimension 4.

Algo.	λ_exp	massl	mnlv	ovo
GO	10.0	0	1	0
RO	12.32..	0	1	0
BO	6.75..	0	1	0
hpo	8.42..	0	1	0

(c) The best input values found for normal classification on the "Gaussian Quantiles" data set of dimension 2.

Algo.	λ_exp	massl	mnlv	ovo
GO	0.0	0	1	1
RO	4.07..	1	1	1
BO	0.28..	0	1	1
hpo	0.47..	0	1	1

(d) The best input values found for normal classification on the "Gaussian Quantiles" data set of dimension 4.

Figure 5.8: Comparison of optimization algorithms using normal classification on the "Gaussian Quantiles" data sets of dimensions 2 and 4.

It is noteworthy that all four algorithms tend to reach the same best evaluation. The evaluations reached differ significantly for the different data sets, with only an evaluation of 0.48 being reached for "Gaussian Quantiles" of dimension 4 (see Figure 5.8b) while many others reach an evaluation above 0.90. The best evaluations have also been found at relatively different input values overall, suggesting that there are no best settings that apply for normal classification all the time, but that it depends on the data set. For the same data set, however, there are some settings that seem to be particularly good, such as `massl` $= 0$ and `ovo` $= 0$ for the "Circles" data set (Figure 5.5b), and the "Gaussian Quantiles" data set of dimension 2 (Figure 5.8c). For all data sets, `perform_BO` has had to use a random x-value in every step, suggesting that it generally more advisable to use `perform_RO`.

For normal classification, `perform_GO` has also performed well, and in some cases, such as Figure 5.7a, it has even reached an evaluation of above 0.92 before all other algorithms. This may happen if the best configuration is among the first values tried. It should, however, be noted that all other values in the search space will still be evaluated, in case a better value can be found. Lastly, it should be noted that the data set used influences how time-intensive the classification process, and therefore the optimization of its hyper-parameters is. To optimize the hyper-parameters of normal classification on the "Moons" data set, each method has only needed about 5 seconds. For normal classification on the "Blobs" data set of dimension 4, they have taken up to 3 minutes.

5.3 Optimization of Dimension-Wise Classification

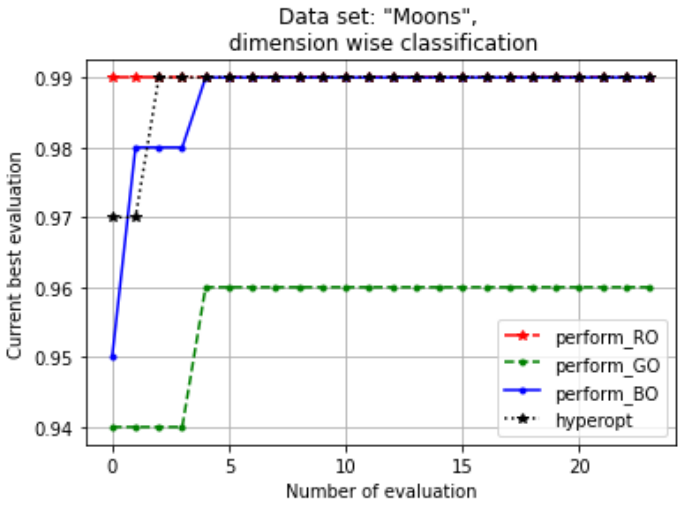
Comparing the optimization methods on dimension-wise classification of different data sets is similar to that of normal classification: An `Optimize_Classification` object is created for each data set, and then `pea_classification_dimension_wise` and `class_dim_wise_space` are used to create a new `HP_Optimization` object. There are also more hyper-parameters that are being optimized. The first three, `lambd_exp`, `massl` and `min_lv` are the same as for normal classification. The fourth, `ovo_ec` is similar but actually encodes two dependent parameters: When `one_vs_others` is activated, one can also choose to use an error calculator for dimension-wise classification. When `ovo_ec` $= 0$, `one_vs_others` is deactivated and therefore no error calculator can be used. When `ovo_ec` $= 1$, `one_vs_others` is activated but no error calculator is used, and when `ovo_ec` $= 2$, an error calculator is used. The last parameter, `max_evaluations`, is an integer between 2 and a parameter `max_evals`, that can be specified when creating an `Optimize_Classification` object and is 256 per default. For the data sets of dimension 4 it is increased to `max_evals` $= 1000$. This parameter controls how often the underlying sparse grid is refined: The sparse grid starts with `min_lv` and `max_lv` and is then refined until the amount of grid points exceeds `max_evaluations`. This also implies that the sparse grid is not refined at all if the first grid used already has more points than `max_evaluations`. As with normal classification, `min_lv` is always 1. The maximal level is set to `max_lv` $= 3$.

Name	Type	Values
lambd_exp	float	[0, 20]
massl	bool	{0, 1}
min_lv	int	{1}
ovo_ec	int	{0, 1, 2}
margin	float	[0, 1]
rebalancing	bool	{0, 1}
use_relative_surplus	bool	{0, 1}
max_evaluations	int	[2, max_evals]

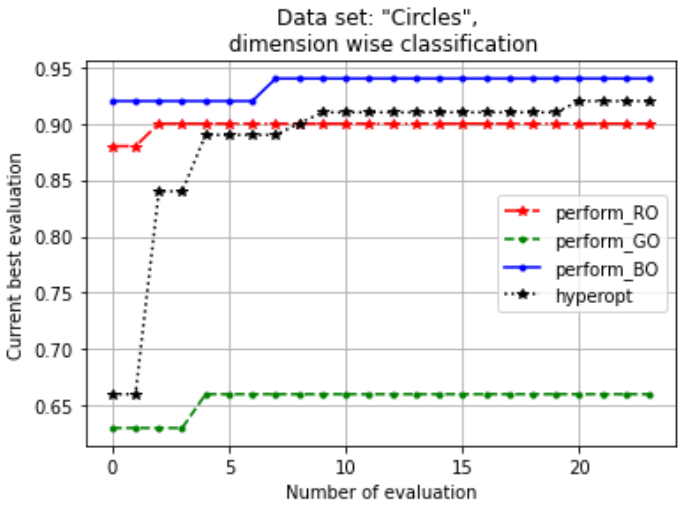
Figure 5.9: The hyper-parameters of dimension-wise classification.

Due to the amount of hyper-parameters, the search space for perform_GO becomes very large. If interval_levels is set to 2 again, this would correspond to a total of 648 evaluations. That is why it is decreased to interval_levels = 0, meaning that only one value is taken from each interval, in this case the smallest. This makes a total of 24 evaluations. As before, perform_RO, perform_BO and hyperopt are set to the same amount of iterations. The data sets on which dimension-wise classification is executed are the same as for normal classification. The comparisons of the optimization algorithms on the different data sets are illustrated by Figs. 5.10 to 5.13.

As with normal classification, perform_BO had to use random values in every step of every optimization process, meaning that it has not performed as intended in every test case. While this may be due to numerical inaccuracies, it may also be due to a bug in my code that has not yet been found. This implies that perform_RO is generally favorable. The first thing that is noteworthy about the comparisons is that perform_GO tends to perform a lot worse than the other algorithms. This is likely because its search space only contains values that perform relatively poorly. The evaluations reached on the same data set are relatively similar again - in some cases with the exception of perform_GO - but they also differ significantly for different data sets. It may be assumed that classification on the data sets where only a low accuracy is reached is generally harder, for both normal and dimension-wise classification. With the exception of the “Classification” data set of dimension 4, and the ”Blobs” data set of dimension 2, dimension-wise classification has reached slightly higher evaluations than normal classification on all data sets. The best values also seem to have changed, as for example the best evaluations for the “Circles” data set do not have massl = 0 and ovo_ec = 0, but they do not have a similar trend either. However, it can also be seen that the best values are not always completely different, as for the “Gaussian Quantiles” data set of dimension 2 all best values still include massl = 0.



(a) Dimension-wise classification on the "Moons" data set.



(b) Dimension-wise classification on the "Circles" data set.

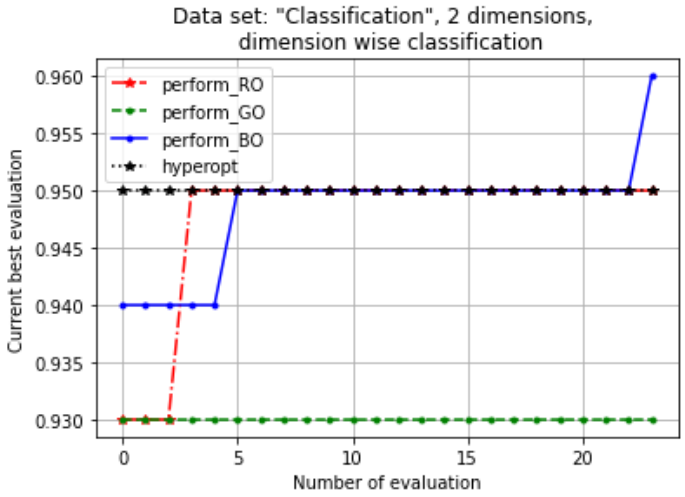
Algo.	Result
GO	(0.0, 0, 1, 1, 0.0, 0, 0, 2)
RO	(6.68.., 0, 1, 2, 0.908.., 1, 0, 15)
BO	(7.84.., 1, 1, 0, 0.298.., 1, 0, 179)
hpo	(7.724.., 1, 1, 2, 0.012.., 1, 1, 220)

(c) The best input values found for dimension-wise classification on the "Moons" data set.

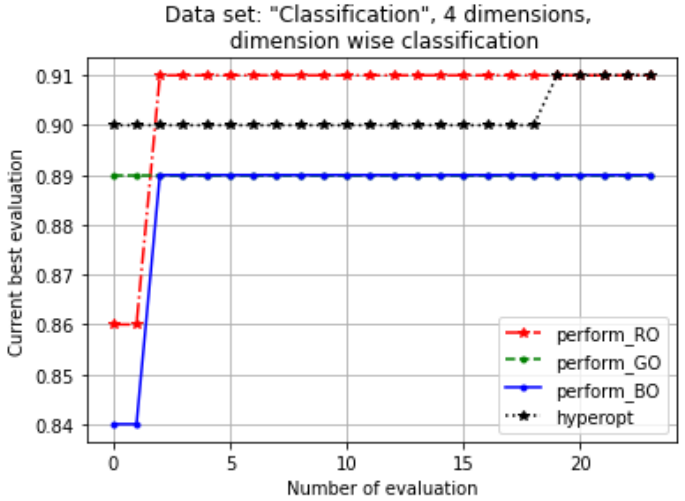
Algo.	Result
GO	(0.0, 0, 1, 1, 0.0, 0, 0, 2)
RO	(5.86.., 1, 1, 1, 0.049.., 0, 0, 205)
BO	(12.114.., 1, 1, 2, 0.08.., 0, 0, 199)
hpo	(3.691.., 0, 1, 0, 0.03.., 0, 0, 201)

(d) The best input values found for dimension-wise classification on the "Circles" data set.

Figure 5.10: Comparison of optimization algorithms using dimension-wise classification on the data sets "Moons" and "Circles". The results are given in the form (`lambd_exp`, `massl`, `min_lv`, `ovo_ec`, `margin`, `rebalancing`, `use_relative_surplus`, `max_evaluations`).



(a) Dimension-wise classification on the "Classification" data set of dimension 2.



(b) Dimension-wise classification on the "Classification" data set of dimension 4.

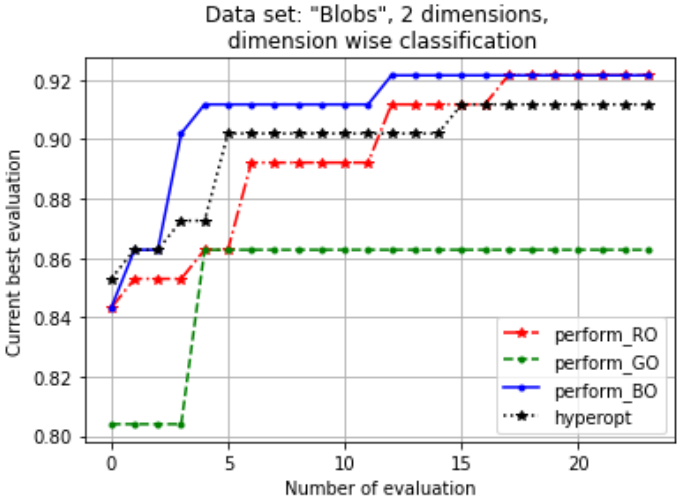
Algo.	Result
GO	(0.0, 0, 1, 0, 0.0, 0, 0, 2)
RO	(12.287.., 1, 1, 2, 0.61.., 1, 1, 183)
BO	(12.323.., 1, 1, 2, 0.273.., 0, 0, 52)
hpo	(15.553.., 0, 1, 1, 0.85.., 1, 1, 51)

(c) The best input values found for dimension-wise classification on the "Classification" data set of dimension 2.

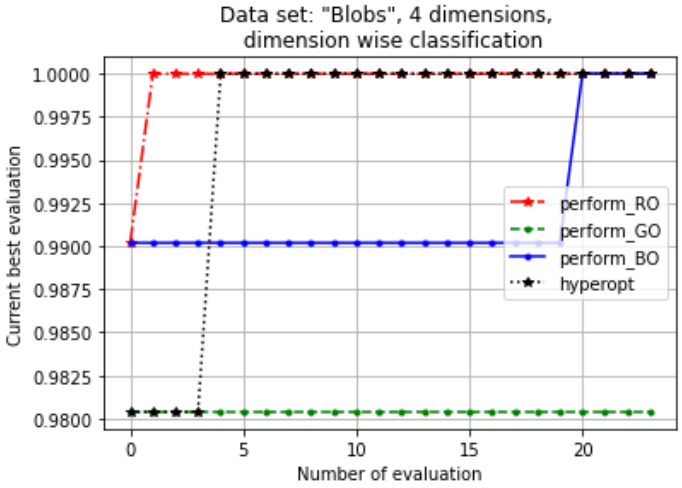
Algo.	Result
GO	(0.0, 0, 1, 0, 0.0, 0, 0, 2)
RO	(19.953.., 1, 1, 2, 0.804.., 0, 0, 683)
BO	(5.157.., 1, 1, 0, 0.604.., 1, 1, 121)
hpo	(7.691.., 1, 1, 1, 0.82.., 1, 0, 216)

(d) The best input values found for dimension-wise classification on the "Classification" data set of dimension 4.

Figure 5.11: Comparison of optimization algorithms using dimension-wise classification on the "Classification" data sets of dimension 2 and 4. The results are given in the form (`lambd_exp`, `massl`, `min_lv`, `ovo_ec`, `margin`, `rebalancing`, `use_relative_surplus`, `max_evaluations`).



(a) Dimension-wise classification on the "Blobs" data set of dimension 2.



(b) Dimension-wise classification on the "Blobs" data set of dimension 4.

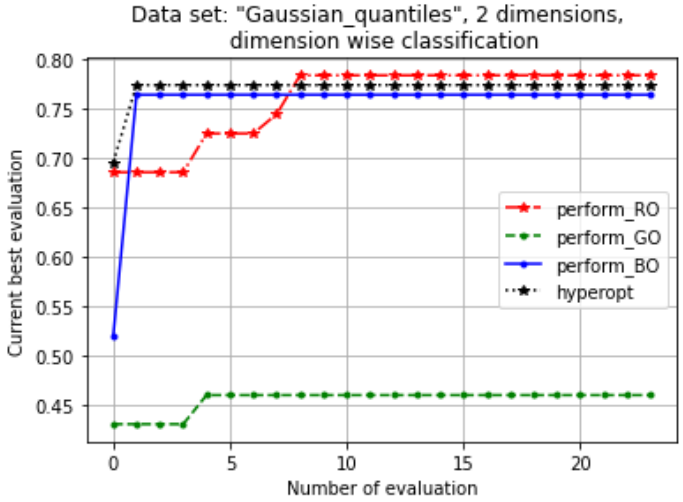
Algo.	Result
GO	(0.0, 0, 1, 1, 0.0, 0, 0, 2)
RO	(4.826.., 1, 1, 2, 0.417.., 0, 0, 215)
BO	(19.875.., 1, 1, 0, 0.753.., 0, 1, 82)
hpo	(7.91.., 0, 1, 1, 0.995.., 1, 1, 101)

(c) The best input values found for dimension-wise classification on the "Blobs" data set of dimension 2.

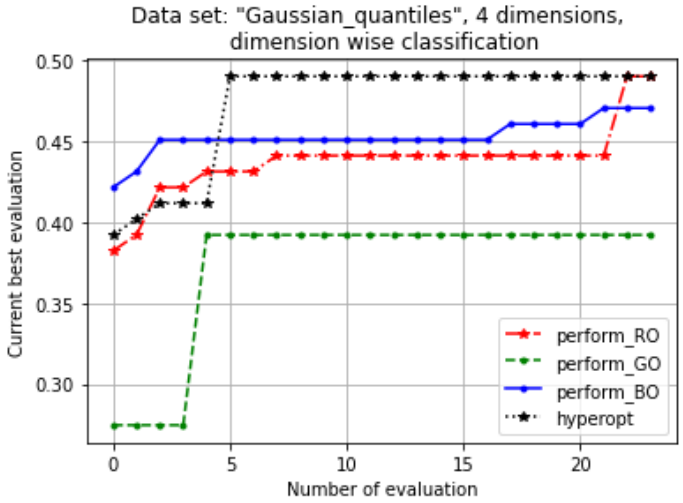
Algo.	Result
GO	(0.0, 0, 1, 0, 0.0, 0, 0, 2)
RO	(15.801.., 0, 1, 2, 0.97.., 1, 0, 743)
BO	(1.916.., 0, 1, 1, 0.706.., 0, 1, 296)
hpo	(6.329.., 1, 1, 2, 0.535.., 1, 1, 133)

(d) The best input values found for dimension-wise classification on the "Blobs" data set of dimension 4.

Figure 5.12: Comparison of optimization algorithms using dimension-wise classification on the "Blobs" data sets of dimension 2 and 4. The results are given in the form (`lambd_exp`, `massl`, `min_lv`, `ovo_ec`, `margin`, `rebalancing`, `use_relative_surplus`, `max_evaluations`).



(a) Dimension-wise classification on the "Gaussian Quantiles" data set of dimension 2.



(b) Dimension-wise classification on the "Gaussian Quantiles" data set of dimension 4.

Algo.	Result
GO	(0.0, 0, 1, 1, 0.0, 0, 0, 2)
RO	(15.447.., 0, 1, 2, 0.475.., 1, 1, 169)
BO	(5.581.., 0, 1, 1, 0.559.., 0, 1, 161)
hpo	(5.391.., 0, 1, 2, 0.558.., 0, 1, 240)

(c) The best input values found for dimension-wise classification on the "Gaussian Quantiles" data set of dimension 2.

Algo.	Result
GO	(0.0, 0, 1, 1, 0.0, 0, 0, 2)
RO	(1.311.., 0, 1, 1, 0.172.., 0, 0, 570)
BO	(17.02.., 1, 1, 2, 0.369.., 1, 1, 755)
hpo	(9.747.., 0, 1, 1, 0.428.., 1, 0, 108)

(d) The best input values found for dimension-wise classification on the "Gaussian Quantiles" data set of dimension 4.

Figure 5.13: Comparison of optimization algorithms using dimension-wise classification on the "Gaussian Quantiles" data sets of dimension 2 and 4. The results are given in the form (`lambd_exp`, `massl`, `min_lv`, `ovo_ec`, `margin`, `rebalancing`, `use_relative_surplus`, `max_evaluations`).

6 Conclusion

"How can the parameters of a given machine learning algorithm be optimized?". In order to discuss this question, this thesis has first introduced the concept of density estimation, a tool that estimates the underlying density function of a given data set. Then, two common machine learning algorithms, classification and clustering, have been presented. Classification is given a data set where each data point is labeled with a "class", and aims to understand the structure of these classes. Clustering operates on a set of unlabeled data and searches for "interesting structure" in it. Both can be implemented based on density estimation.

It has then been explained that for machine learning algorithms, some parameters need to be set before the learning process. These are called hyper-parameters. Optimizing them is a non-trivial task, and doing so manually requires expert knowledge that is often not available. That is why hyper-parameter optimization methods aim to automatize this process. Mathematically, they maximize an objective function that maps a given set of hyper-parameters to an evaluation. Three common methods for this are grid search, random search and Bayesian optimization. Grid Search is a type of exhaustive search, where all values specified in a search space are evaluated to find the best one. Random search searches through random values, while Bayesian optimization aims to learn from former evaluations in order to estimate what values may be good.

These three methods have been implemented in the scope of this thesis. They, as well as the open source software hyperopt, have then been used to optimize the hyper-parameters of normal and dimension-wise classification, two classification algorithms that have been previously implemented on the sparseSpACE framework. These tests have shown that the performance depends strongly on the data set on which classification is performed. However, in most cases a relatively good configuration with an accuracy of above 90% has been obtained. It was also shown that the performance of grid search worsens when there are more hyper-parameters: For normal classification, which has four hyper-parameters, grid search usually reached an equally high evaluation as the other methods, while it was significantly worse for dimension-wise classification, which has eight hyper-parameters. This is likely because the search-space used for dimension-wise classification included only few values for each hyper-parameter, and probably left out other values that would have produced a better performance. If more values per hyper-parameter are included, however, the size of the search space grows considerably, and thus the computational cost. Lastly, my implementation of Bayesian optimization was shown to be flawed, as it always resorts to evaluating random values (for more information on this, please refer to Subsection 4.2.1). This means it performs similarly to random search, but requires more computational effort.

Therefore, future research could be dedicated to understanding this flaw and improving on it. Additionally, other hyper-parameter optimization methods may be implemented, such as population based methods and different types of Bayesian optimization.

List of Figures

Bibliography

[1] K. P. Murphy. *Machine learning: a probabilistic perspective*. Cambridge, MA, 2012.

[2] T. Yu and H. Zhu. *Hyper-Parameter Optimization: A Review of Algorithms and Applications*. 2020. arXiv: 2003.05689 [cs.LG].

[3] B. Peherstorfer, D. Pflüge, and H.-J. Bungartz. "Density Estimation with Adaptive Sparse Grids for Large Data Sets". In: *Proceedings of the 2014 SIAM International Conference on Data Mining (SDM)*, pp. 443–451. DOI: 10.1137/1.9781611973440.51. URL: https://epubs.siam.org/doi/abs/10.1137/1.9781611973440.51.

[4] H.-J. Bungartz and M. Griebel. "Sparse grids". In: *Acta Numerica* 13 (2004), pp. 147–269. ISSN: 1474-0508. DOI: 10.1017/S0962492904000182.

[5] M. Griebel, M. Schneider, and C. Zenger. "A combination technique for the solution of sparse grid problems". In: *Iterative Methods in Lin. Alg.* 1992, pp. 263–281.

[6] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. "Scikit-Learn: Machine Learning in Python". In: *J. Mach. Learn. Res.* 12.null (Nov. 2011), 2825–2830. ISSN: 1532-4435.

[7] C. C. Moser. "Machine Learning with the Sparse Grid Density Estimation using the Combination Technique". Bachelor's Thesis. Technical University of Munich, 2020.

[8] B. Peherstorfer, F. Franzelin, D. Pflüger, and H.-J. Bungartz. "Classification with Probability Density Estimation on Sparse Grids". In: *Sparse Grids and Applications - Munich 2012*. Ed. by J. Garcke and D. Pflüger. Cham: Springer International Publishing, 2014, pp. 255–270. ISBN: 978-3-319-04537-5.

[9] B. Peherstorfer, D. Pflüger, and H.-J. Bungartz. "Clustering Based on Density Estimation with Sparse Grids". In: *KI 2012: Advances in Artificial Intelligence*. Ed. by B. Glimm and A. Krüger. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 131–142. ISBN: 978-3-642-33347-7.

[10] K. Maladkar. *Why Is Random Search Better Than Grid Search For Machine Learning*. 2018. URL: https://analyticsindiamag.com/why-is-random-search-better-than-grid-search-for-machine-learning/ (visited on 08/28/2021).

[11] D. Shiffman, S. Fry, and Z. Marsh. *The Nature of Code*. D. Shiffman, 2012. ISBN: 9780985930806. URL: https://natureofcode.com/book/.

[12] 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.

[13] W. Koehrsen. *A Conceptual Explanation of Bayesian Hyperparameter Optimization for Machine Learning*. 2018. URL: `https://towardsdatascience.com/a-conceptual-explanation-of-bayesian-model-based-hyperparameter-optimization-for-machine-learning-b8172278050f` (visited on 08/28/2021).

[14] P. Luong, S. Gupta, D. Nguyen, S. Rana, and S. Venkatesh. "Bayesian Optimization with Discrete Variables". In: *AI 2019: Advances in Artificial Intelligence*. Ed. by J. Liu and J. Bailey. Cham: Springer International Publishing, 2019, pp. 473–484. ISBN: 978-3-030-35288-2.

[15] C. E. Rasmussen and C. K. I. Williams. *Gaussian Processes for Machine Learning*. MIT Press, 2006.

[16] N. Srinivas, A. Krause, S. M. Kakade, and M. W. Seeger. "Information-Theoretic Regret Bounds for Gaussian Process Optimization in the Bandit Setting". In: *IEEE Transactions on Information Theory* 58.5 (2012), 3250–3265. ISSN: 1557-9654. DOI: `10.1109/tit.2011.2182033`. URL: `http://dx.doi.org/10.1109/TIT.2011.2182033`.

[17] M. Obersteiner. *sparseSpACE - The Sparse Grid Spatially Adaptive Combination Environment*. 2007.

[18] L. Schulte. "Sparse Grid Density Estimation with the Combination Technique". Bachelor's Thesis. Technical University of Munich, 2020.

[19] M. Fabry. "Spatially adaptive Density Estimation with the Sparse Grid Combination Technique". Master's Thesis. Technical University of Munich, 2020.

[20] J. Bergstra. *Hyperopt: Distributed Hyperparameter Optimization*. 2013. URL: `https://github.com/hyperopt/hyperopt` (visited on 09/07/2021).

[21] J. Bergstra, D. Yamins, and D. D. Cox. "Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures". In: *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*. ICML'13. Atlanta, GA, USA: JMLR.org, 2013, I–115–I–123.