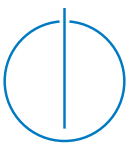
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Adaptive Regression with the Spatially Adaptive Combination Technique

Maximilian Michallik



DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Adaptive Regression with the Spatially Adaptive Combination Technique

Adaptive Regression mit der räumlich-adaptiven Kombinationstechnik

Author:	Maximilian Michallik
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	M.Sc. Michael Obersteiner
Submission Date:	15.08.2021

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

Munich, 15.08.2021 Maximilian Michallik

Acknowledgments

I would like to thank my advisor, Michael Obersteiner, for his advice and support throughout the regular meetings. Thank you for your feedback and your suggestions, they helped me a lot!

I also want to thank my family, my girlfriend, and my friends for the continuous support.

Abstract

The complexity of reconstructing a function in high dimensions is too complex for modern computers with regular grids. This *curse of dimensionality* can be tackled with the Sparse Grid Combination Technique. By reducing the number of grid points, the complexity is reduced while still having accurate results.
In this thesis, regression with the spatially adaptive Combination Technique is implemented for different execution options. We introduce and analyze two approaches for regularization, which tackle the problem of overfitting. Additionally, we implemented three different versions for Opticom which update the coefficients of the component grids according to different criteria as additional optimizations. The various execution options which are implemented in the sparseSpACE[1] framework are compared with each other regarding accuracy and complexity. Moreover, we compare the implementation to other common regression approaches. The results show that we can outperform neural networks and polynomial models in certain cases.

[1] https://github.com/obersteiner/sparseSpACE

Zusammenfassung

Die Komplexität der Rekonstruktion einer Funktion in hohen Dimensionen ist für moderne Computer mit regelmäßigen Gittern zu komplex. Dieser *Fluch der Dimensionalität* kann mit der Sparse-Grid-Combination-Technique angegangen werden. Durch die reduzierte Anzahl von Gitterpunkten wird die Komplexität reduziert, während dennoch genaue Ergebnisse erzielt werden.
In dieser Arbeit wird die Regression mit der räumlich-adaptiven Kombinationstechnik mit verschiedenen Ausführungsoptionen implementiert. Zwei Ansätze zur Regularisierung, die das Problem der Überanpassung angehen, werden vorgestellt und evaluiert. Zusätzlich kommen drei verschiedene Versionen für Opticom hinzu, die als zusätzliche Optimierungen die Koeffizienten der Komponentengitter nach unterschiedlichen Kriterien aktualisieren. Die verschiedenen Ausführungsmöglichkeiten, die im sparseSpACE[2] Framework implementiert sind, werden hinsichtlich Genauigkeit und Komplexität miteinander verglichen. Es werden zusätzliche Tests durchgeführt, die die Implementierung mit anderen gängigen Regressionsmethoden vergleichen. Die Ergebnisse zeigen, dass die Vorhersagen von Testdaten in bestimmten Fällen besser sind als neuronale Netze.

[2]https://github.com/obersteiner/sparseSpACE

Contents

1 Introduction

With the increasing amount of data that is being gathered, the difficulty of extracting information gets harder. Especially for humans, it is not easy to handle data sets in a meaningful way. Therefore computers have to be used to automate this. There are different tasks that are of interest when handling big data. They all have one thing in common, namely that they are usually more expensive for higher dimensionalities. Unfortunately, many data sets are high-dimensional in practice. This is the reason why a solution that can handle this *curse of dimensionality* is required.

A remedy for this problem is to use *Sparse Grids* [1]. The idea of this approach is to reduce the number of grid points which then leads to faster computation times. Especially at regions where fewer samples are located, points can be left out without sacrificing accuracy. This can be done in a static way or by refining towards the data points.

One task when dealing with a huge amount of data is regression which will be the focus of the next chapters. Given a data set with many samples with each of them having a so called target value, the goal is to reconstruct the function that generated those samples. To achieve this goal of regression, the *sparseSpACE* framework will be used that implements the most important operations for Sparse Grids.

After the theoretical background in chapter 2, where Sparse Grids, regression and Opticom are introduced, the implementation is presented in chapter 3. There, we present the framework used and the added functionality. All different execution options are described. In the following chapter 4, we present the experiments and evaluate certain properties of the adaptive regression with the spatially adaptive Combination Technique. We take a deeper look into different aspects of the implementation and compare it with other common regression methods. In the last chapter 5, the conclusion is drawn and we present ideas on how to further improve the implementation.

2 Theoretical Background

In this chapter, the theoretical background of the implementation is explained. First, full grids and the numerical approximation of functions are explained in section 2.1. Then, Sparse Grids are introduced in 2.2 as a solution of applications in high dimensions. Especially the Combination Technique and adaptivity are explained. Then, the goal of regression is stated in 2.3 with the mathematical background and in the end, the approach of optimizing regression in the context of Sparse Grids with Opticom will be explained in section 2.4. The following is mainly based on [2] if not stated otherwise. For further readings on the numerical approximation of functions and Sparse Grids, refer to [1], [3].

2.1 Numerical approximation of functions

Consider a function $f : \Omega \to \mathbb{R}$ with $\Omega = [0,1]^d$ being the unit interval in d dimensions. The classical approach is to represent f using a full grid with n grid points in each dimension. Altogether this leads to n^d points which is usually only practicable to a dimension up to 4 [2, page 5]. That is the reason why Sparse Grids will be introduced later in this chapter as a solution for the problem of the curse of dimensionality.

In the beginning, we present the base case with $d = 1$, the step into multiple dimensions will be explained later in this section. The discretization level l determines the number of grid points in the interval $[0,1]$. This results in a mesh with $2^l - 1$ grid points which have a distance of $h_l = 2^{-l}$ to each neighbor. Altogether the points with indices in $I_l := [1, 2^l - 1]$ are

$$x_{l,i} := i \cdot h_l, i \in I_l. \tag{2.1}$$

For the sake of simplicity, the boundary treatment will not be discussed.

Now with this grid of level l, a function can be represented as a weighted sum of basis functions

$$f_l(x) = \sum_{i \in I_l} \alpha_{l,i} \varphi_{l,i}(x) \tag{2.2}$$

which are combined either resulting in the nodal basis or the hierarchical basis. We are using the first variant now.

The basis functions $\varphi_{l,i}$ are centered on the $x_{l,i}$

$$\varphi_{l,i}(x) = \varphi(\frac{x - i \cdot h_l}{h_l}) \tag{2.3}$$

and have the support $[x_{l,i} - h_l, x_{l,i} + h_l]$. One possible choice for the basis is the hat function (see also Figure 2.1) which are combined to form the nodal basis.

$$\Phi(x) = \max(1 - |x|, 0) \tag{2.4}$$

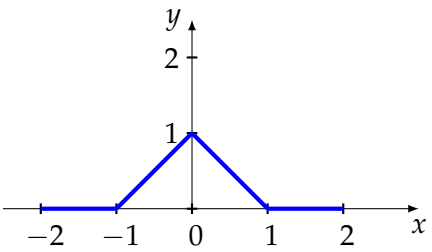


Figure 2.1: Hat function Φ in the interval [-2,2]

An example of a representation as a weighted sum can be seen in Figure 2.2 where the discretization level of the grid is 3. The boundary is considered to be zero in this example.

The second basis representation is the hierarchical basis. Here, the index set is $I_l^h := \{i \in \mathbb{N} | 1 \leq i \leq 2^l - 1,\ i \text{ odd}\}$. This leads to the set of hierarchical subspaces

$$W_l := \text{span}\{\varphi_{l,i}(x) | i \in I_l^h\} \tag{2.5}$$

By combining those subspaces in a direct sum

$$V_n = \bigoplus_{l \leq n} W_l \tag{2.6}$$

the whole space of piecewise linear functions V_n can be obtained (see also Figure 2.3). Each W_i alone can not represent any arbitrary function because they are always zero at certain points. But together, they can represent all functions in V_3.

Similar to the nodal basis, a function can be represented with the hierarchical basis as a weighted sum of the basis functions:

$$f_l(x) = \sum_{i \in I_l^h} \alpha_{l,i} \varphi_{l,i}(x) \tag{2.7}$$

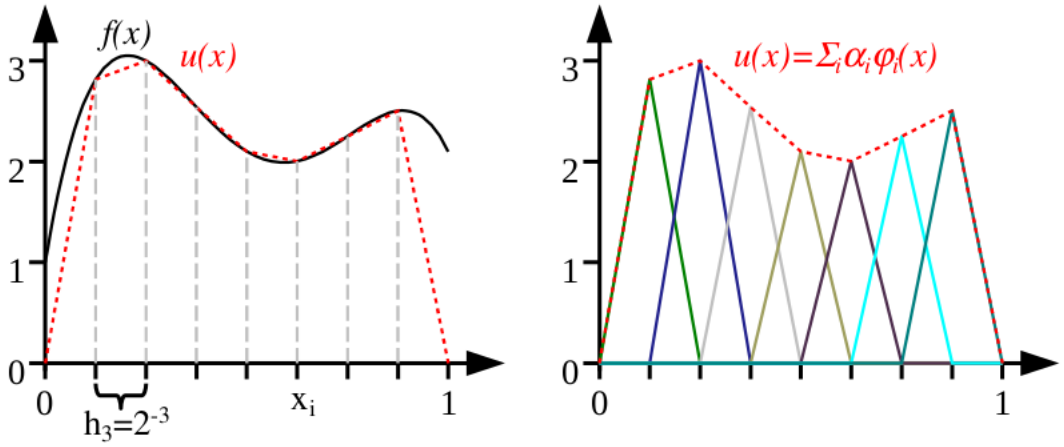


Figure 2.2: Piecewise linear interpolation $u(x)$ (dashed) of $f(x)$ (solid) using a grid of level 3. Combination of the hat functions can be seen on the right, taken from [2].

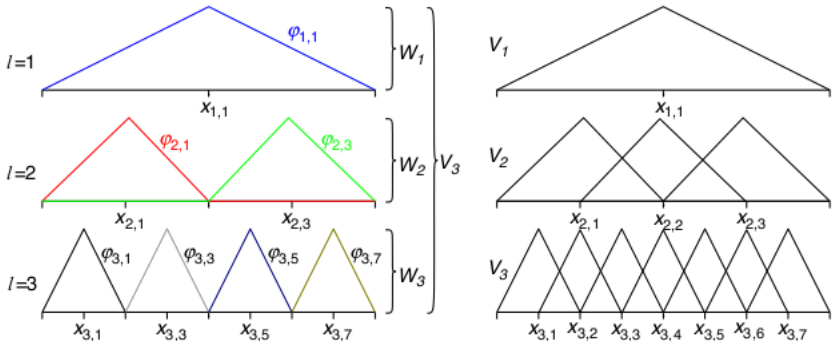


Figure 2.3: Comparison of the hierarchical basis (left) and the nodal basis (right). Basis functions $\varphi_{l,i}$ and grid points $x_{l,i}$, taken from [2].

The same example as for the nodal basis can be seen in Figure 2.4. This time, the hierarchical basis is used.

To extend this to an arbitrary dimension d, a tensor product based approach is used.

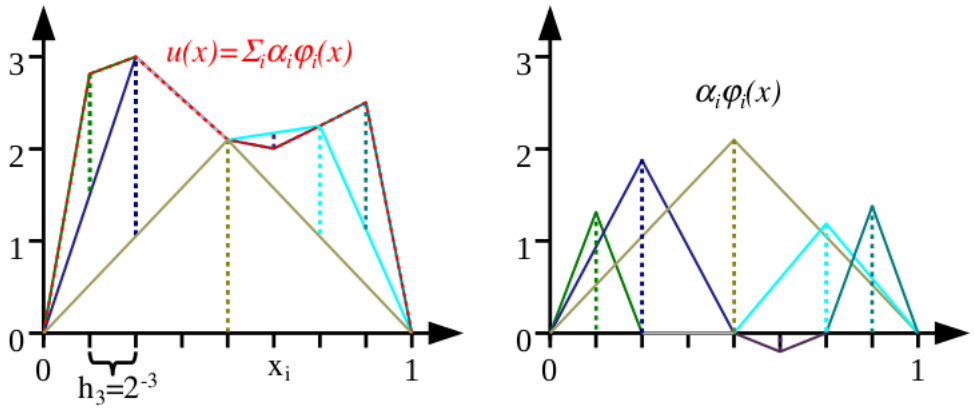


Figure 2.4: Piecewise linear interpolation using the hierarchical basis of same function as in 2.2 (left), basis functions with height α_i (right), taken from [2].

The basis functions φ are now

$$\varphi_{\vec{l},\vec{i}}(\vec{x}) = \prod_{j=1}^{d} \varphi_{l_j,i_j}(x_j) \tag{2.8}$$

with $\vec{l}$ and $\vec{i}$ indicating level and index for each dimension. The index set is now

$$I_{\vec{l}} := \{\vec{i} \,|1 \leq i_j \leq 2^{l_j} - 1,\ i_j \text{ odd}, 1 \leq j \leq d\} \tag{2.9}$$

and the subspaces $W_{\vec{l}}$

$$W_{\vec{l}} := \text{span}\{\varphi_{\vec{l},\vec{i}}(\vec{x}) | \vec{i} \in I_{\vec{l}}\}. \tag{2.10}$$

All in all, the function space V_n can be constructed by

$$V_n = \bigoplus_{|\vec{l}|_\infty \leq n} W_l, \tag{2.11}$$

with $|\vec{l}|_\infty := \max_{1 \leq i \leq d} |d_i|$. With every dimension having $2^n - 1$ grid points, this leads to an overall number of $(2^n - 1)^d$ of points. Now a function can be represented as

$$f(\vec{x}) = \sum_{|\vec{l}|_\infty \leq n, \vec{i} \in I_{\vec{l}}} \alpha_{\vec{l},\vec{i}} \varphi_{\vec{l},\vec{i}}(\vec{x}). \tag{2.12}$$

The subspaces in two dimensions can be seen in Figure 2.5, where all basis functions are created with the tensor product of the 1D hat functions.

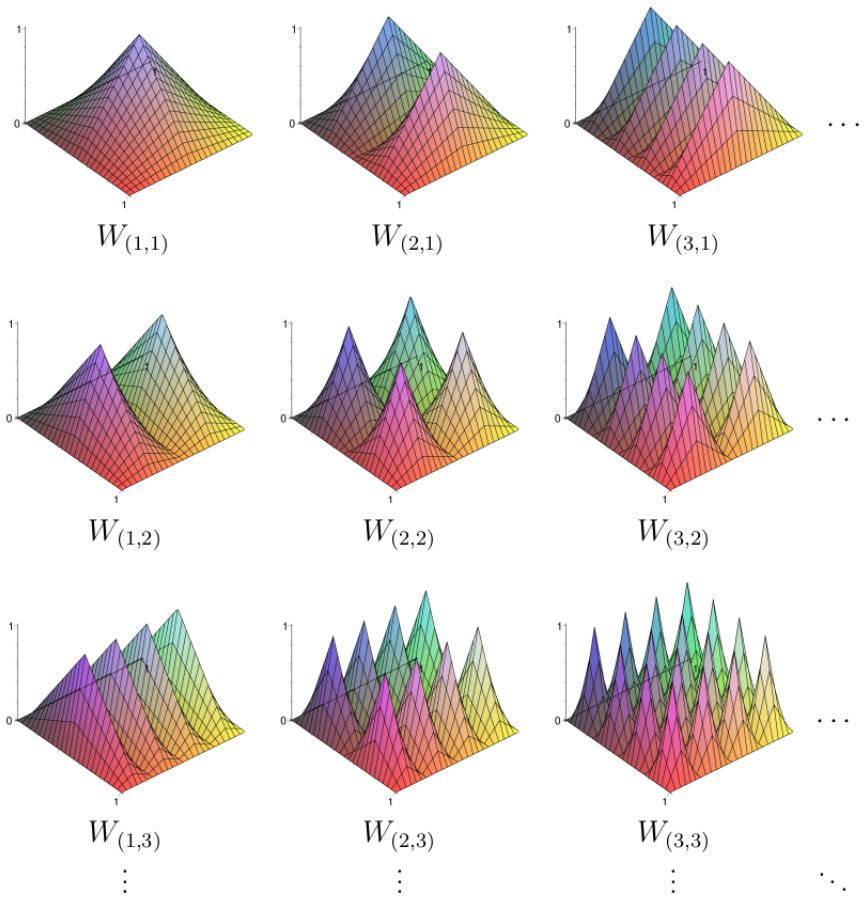


Figure 2.5: Two dimensional subspaces with the contained basis functions, taken from [2].

This interpolation of the function in a regular grid leads to an overall error of

$$||f(\vec{x}) - u(\vec{x})||_{L_2} \in \mathcal{O}(h_n^2) \tag{2.13}$$

with a cost of

$$\mathcal{O}(h_n^{-d}) = \mathcal{O}(2^{nd}) \tag{2.14}$$

function evaluations [1, pages 17-18]

2.2 Sparse Grids

The problem of regular full grids is the *curse of the dimensionality* meaning that the complexity of a problem scales exponentially with its dimension. The number of grid points and therefore the number of function evaluations of a full grid with dimension d and number of grid points n in each dimension leads to n^d points. A solution to tackle this problem is the use of Sparse Grids. The idea is to reduce the number of grid points leading to reduced complexity with still high accuracy. There are different approaches of building a Sparse Grid. Two of them will be presented. The first one being the normal approach followed by the Combination Technique which is also used in the implementation.

The difference of the first approach (normal way of constructing Sparse Grids) to the full grid approach is to leave out some subspaces from the full grid which contribute little to the overall accuracy. The Sparse Grid space is then

$$V_n^1 = \bigoplus_{|\vec{l}|_1 \leq n+d-1} W_{\vec{l}}, \tag{2.15}$$

and a function in this space can be formulated as

$$f(\vec{x}) = \sum_{|\vec{l}|_1 \leq n+d-1, \vec{i} \in I_{\vec{l}}} \alpha_{\vec{l},\vec{i}} \varphi_{\vec{l},\vec{i}}(\vec{x}). \tag{2.16}$$

An example for $n = 3$ can be seen in Figure 2.6. The resulting grid has significantly less grid points. In comparison to the full grid of the same level (see also 2.14), Sparse Grids only have

$$\mathcal{O}(h_n^{-1}(\log h_n^{-1})^{d-1}) \text{ points}, \tag{2.17}$$

while the accuracy only slightly reduces to

$$\mathcal{O}(h_n^2(\log h_n^{-1})^{d-1}), \tag{2.18}$$

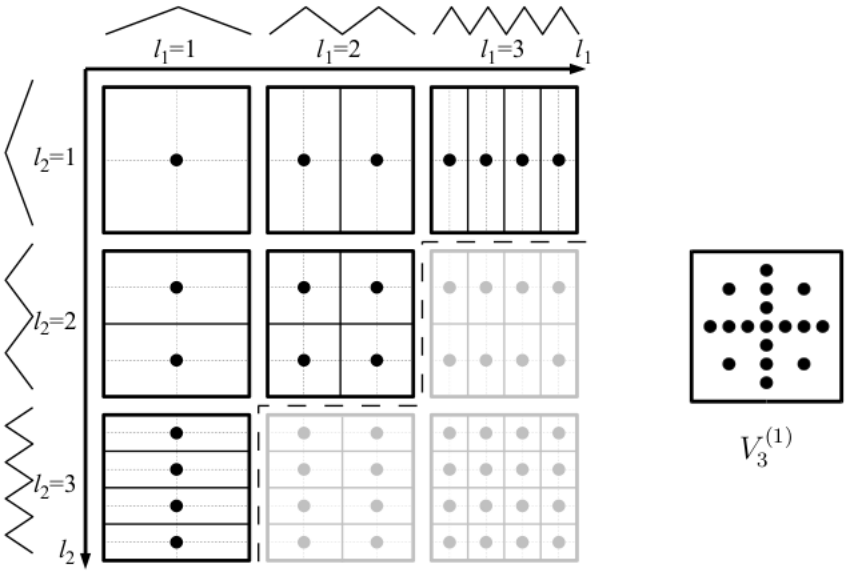


Figure 2.6: Subspaces in two dimensions and the resulting Sparse Grid, taken from [2].

(see equations 2.13 and 2.14 for comparison) [2, page 12].

Another approach, which is different to the first one, is to use the Combination Technique [4], [5]. The idea is to combine different anisotropic full grids by addition and subtraction to get a Sparse Grid. Those grids Ω_l have uniform mesh sizes $h_t = 2^{-l_t}$ in the t-th coordinate direction. The grids that are important for the combination technique have

$$l_1 + ... + l_d = n + (d-1) - q,\ q = 0, ..., d-1, l_t > 0. \tag{2.19}$$

The different grids and the resulting Sparse Grid for level 5 are shown in Figure 2.7.

An interpolated function would then be given by adding all weights of the basis functions of the green grids and subtracting those from the orange grids

$$f(\vec{x}) = \sum_{k=0}^{d-1} (-1)^k \binom{d-1}{k} \sum_{|\vec{l}|_1 \leq n+(d-1)-k} f_{\vec{l}}(\vec{x}). \tag{2.20}$$

In this case with $n = 5$ and dimension 2, the ones with $|\vec{l}|_2 = 6$ are added (green) and the orange grids with $|\vec{l}|_2 = 5$ are subtracted. This is necessary because we have to remove duplicated points. This results in an overall number of

$$\mathcal{O}(d(\log h_n^{-1})^{d-1}) \times \mathcal{O}(h_n^{-1}) \text{ grid points} \tag{2.21}$$

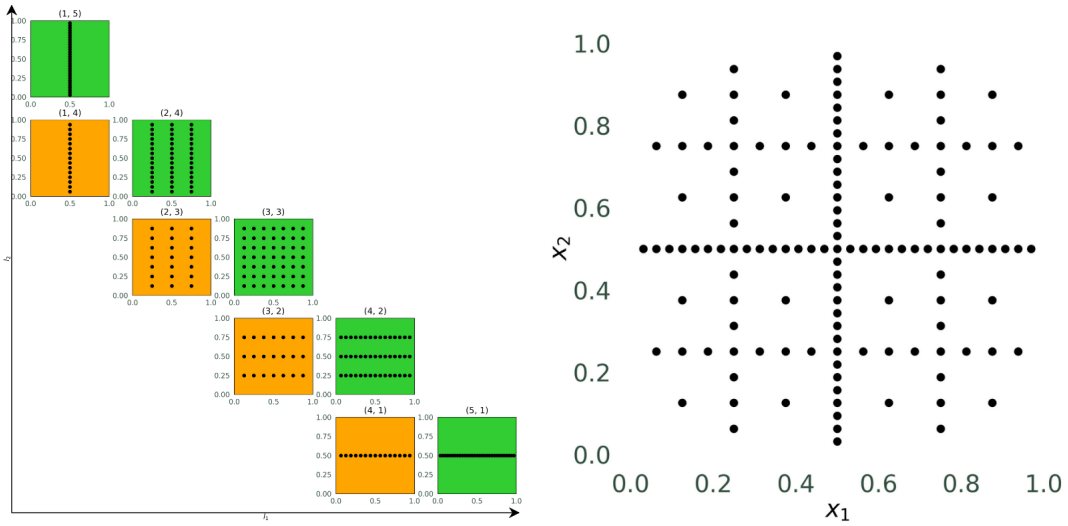


Figure 2.7: Example for a combination scheme with minimum level 1 and maximum level 5. The green component grids are added and the orange ones are subtracted. On the right, the resulting Sparse Grid.

and an error of

$$\mathcal{O}(h_n^2(\log h_n^{-1})^{d-1}) \tag{2.22}$$

[2, page 20].

An addition to this approach is to use refinement. The idea is to adapt the grid to be asymmetric so that it fits the data set well. Local error estimations indicate where the grid has to be refined. There, the hierarchical children of the local grid point are also added to the grid (see Figure 2.8). In the Figure, the red grid points are refined leading

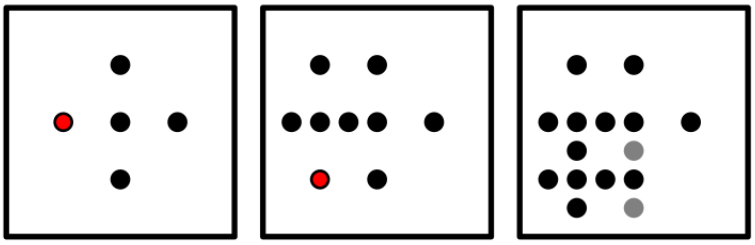


Figure 2.8: Example for adaptivity. The red points are refined and the grey points have to be added because they are the hierarchical parents of the added points, taken from [2].

to a higher density of the grid in the bottom left corner. The gray grid points of the right grid had to be added recursively because they represent hierarchical parents of the newly added grid points. Those hierarchical parents are usually needed to transform the grid values to the hierarchical basis.

The grid is then refined in several steps. In each step, the error in the different regions is computed. At those grid points where the current error is still high, refinement should target this error. A more detailed description can be found in [6]. This is done until a stopping criterion is fulfilled. An example of this is a limit of the overall number of grid points or a tolerance of the error. An example for such a refinement with the Combination Technique can be seen in figure 2.9.

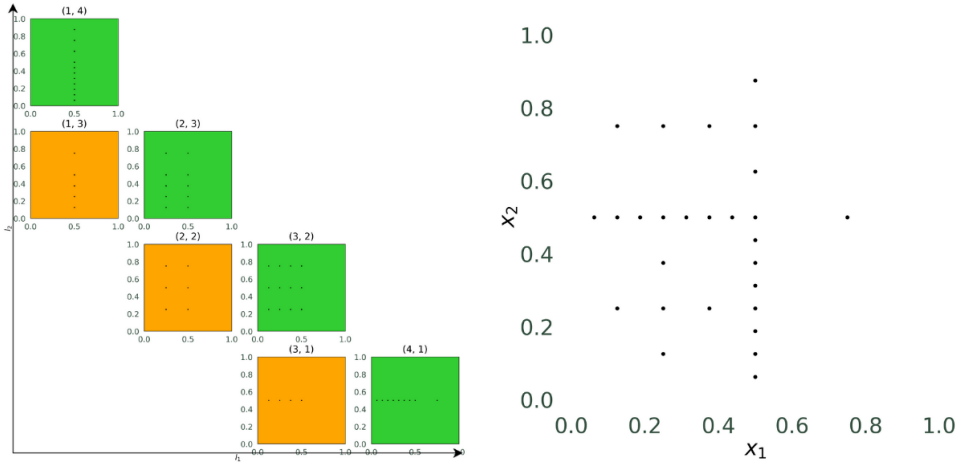


Figure 2.9: Example for the spatially adaptive Combination Technique. The scheme with the component grids on the left and the resulting Sparse Grid on the right. The green ones are added and the orange grids are subtracted.

The function and the samples are depicted in figure 2.10. The maximum of this function is at $(0.2, 0.3)$ and around this point, the error is still high. That is the reason why the grid is refined in this region.

2.3 Regression with Sparse Grids

Given a data set S of m data points of dimension d with target values

$$S = \{(\vec{x}_i, y_i) \in \mathbb{R}^d \times \mathbb{R}\}_{i=1}^{m}, \tag{2.23}$$

which are samples of a specific function f with possible noise on a certain grid with specified basis functions, the goal of regression is to reconstruct f. In the case of Sparse

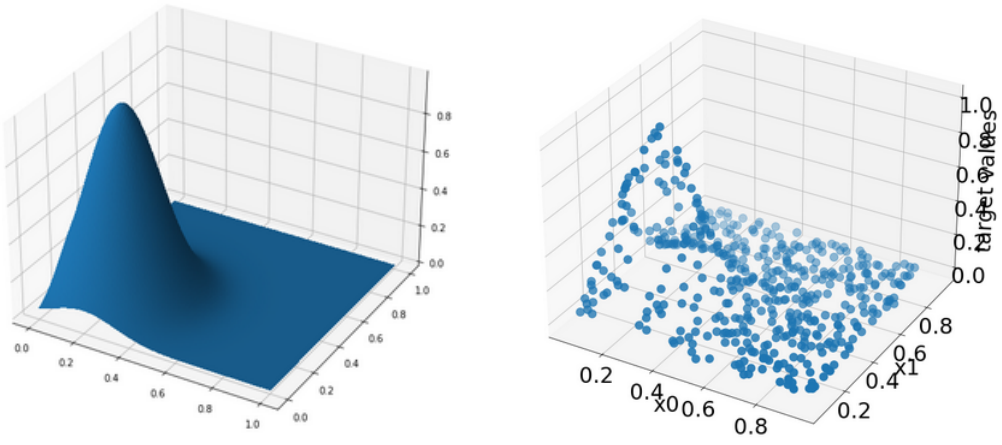


Figure 2.10: Example function (Gaussian function) as a plot on the left and data samples on the right.

Grids, the domains of the data points can be adapted to be $\Omega = [0,1]^d$ by scaling. The following is mainly based on [4] if not stated otherwise.

A first simple approach to solve this problem is to use the basis functions φ of the function space in the following way. With the matrix A and the vector $\vec{y}$, the coefficients $\vec{\alpha}$ for the φ can be found by solving

$$\arg\min_{\vec{\alpha}} |A\vec{\alpha} - \vec{y}| \tag{2.24}$$

with a least squares approach. Here, $A_{i,j} = \varphi_j(\vec{x}_i)$ and $\vec{y}$ denotes the vector of all target values. With $A \in \mathbb{R}^{m \times d}$ (with m number of data points and d dimension and $d << m$), this is an overdetermined system of linear equations. This can be solved by least squares. The goal is to minimize the sum of the squared residuals

$$S = \sum_{i=1}^{m} r_i^2 \text{ , with } r_i = y_i - \sum_{j=1}^{d} \alpha_j \phi_j(x_i). \tag{2.25}$$

The approximation u of f is then the sum of the basis functions which are weighted according to $\vec{\alpha}$. Until now, the reconstructed function is only based on the data points. This can lead to overfitting if the solution that is found is too exact. To overcome this problem, we present a refined approach with regularization. This tackles the problem of overfitting by making the reconstructed function smooth.

By adding a regularization term to this regression problem, the minimization

$$u = \arg\min_{u \in V} (\frac{1}{m} \sum_{i=1}^{m} (y_i - u(\vec{x}_i))^2 + \lambda C(u)) \tag{2.26}$$

with V being a function space over Ω and a regularization functional $C(u)$ results. The problem consists of two parts. The first term ensures that the approximation u is similar to f. The second term is used for regularization. It prevents overfitting by penalizing non-smooth function shapes. With the regularization parameter λ, which is dependent on the application data set, a trade off between exactness and smoothness can be determined. For this regularization, we are using two different approaches.

One first approach for C is

$$C(u) := ||\nabla f||_{L_2}^2 \tag{2.27}$$

with $||f(\vec{x})||_{L_2}^2 = \int_\Omega f^2 d\vec{x}$. Another possibility is to use the Euclidean norm of the vector of surpluses in case of using Sparse Grids

$$C(u) := ||\vec{\alpha}||_2^2. \tag{2.28}$$

By minimizing 2.26 with regularization functional 2.28, the linear system

$$(\frac{1}{m} B B^T + \lambda I)\vec{\alpha} = \frac{1}{m} B\vec{y} \tag{2.29}$$

has to be solved with $B = A^T$ (with A from the first approach), I being the identity matrix and the vector $\vec{y}$ the target values. This is also a least squares problem, this time with a regularization term.

The choice of the second regularization functional (2.28) is an approximation of the solution with 2.27 instead. This can be useful as the computational cost is lower than it is for using 2.27. Although, the results of some applications are less accurate and the failure rates can be higher. So with the regularization functional 2.27, the minimization of 2.26 leads to the system of linear equations

$$(\frac{1}{m} B B^T + \lambda C)\vec{\alpha} = \frac{1}{m} B\vec{y}. \tag{2.30}$$

Note that 2.29 and 2.30 only differ in the matrix of the regularization functional. In the more accurate case, the matrix consists of $(C)_{i,j} = \langle \nabla \varphi_i, \nabla \varphi_j \rangle_{L_2}$.

Now in the case of using the Combination Technique, the regression is solved independently on each component grid. The resulting functions are then combined to the overall solution. An example can be seen in Figure 2.11. The functions of the component grids (1,4), (2,3), (3,2), (4,1) are simply added and the others are subtracted.

2.4 Opticom

The result being a combination of partial solutions can now still be optimized. The classical Combination Technique combines the component grids with weights that are

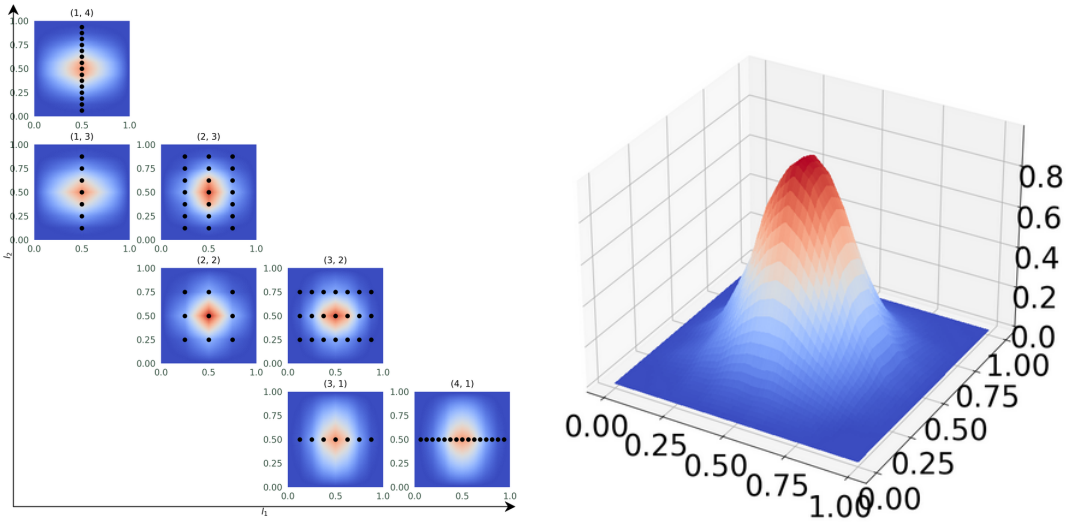


Figure 2.11: Results of the regression on the component grids on the left. The results of grids (1,4), (2,3), (3,2) and (4,1) are added, the others are subtracted. The resulting overall solution is on the right.

integers. This can be improved by finding better coefficients of the single grids that are combined. The resulting weights do not necessarily have to be integers. This results in the Opticom that was introduced in [7]. In the scope of this paper, three approaches are implemented and tested.

For the first approach, a linear system is solved for the coefficients $\vec{c}$. The idea is based on the fact that for each point $\vec{x}$, the resulting target point can be computed as weighted sum

$$y = \sum_{i=1}^{k} c_i u_i(\vec{x}) \tag{2.31}$$

with c_i being the weight or coefficient of the i^{th} component grid and $u_i(\vec{x})$ the predicted target value of the point $\vec{x}$ on this grid.

To optimize those coefficients so that the function is reconstructed as good as possible, this equation has to hold true for all available data points. This leads to the problem

$$\arg\min_{\vec{c}} |A\vec{c} - \vec{y}| \tag{2.32}$$

with $\vec{y}$ being the vector with all available target values and $A_{i,j} = u_i(x_j)$ (from the i^{th} component grid). We can solve it using a least squares approach. No smoothness is guaranteed in this variant of Opticom which possibly leads to an overfitted solution

because we are only considering the data points that are available at this time (training and validation set).

In contrast, the second approach introduced by [8] has a regularization which guarantees smoothness. This approach also results in a problem that can be solved with a least squares approach with

$$\begin{pmatrix} ||P_1 f||_G^2 & ... & \langle P_1 f, P_m f\rangle_G \\ \langle P_2 f, P_1 f\rangle_G & ... & \langle P_2 f, P_m f\rangle_G \\ ... & ... & ... \\ \langle P_m f, P_1 f\rangle_G & ... & ||P_m f||_G^2 \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ ... \\ c_m \end{pmatrix} = \begin{pmatrix} ||P_1 f||_G^2 \\ ||P_2 f||_G^2 \\ ... \\ ||P_m f||_G^2 \end{pmatrix}. \tag{2.33}$$

We denote the function on the component grid i with $P_i f$. The scalar product is defined as $\langle f, g\rangle_G := \langle f, Gg\rangle_2$ with the matrix $G := B^T \cdot B + \lambda M \cdot C$. This leads to the term

$$\langle f, g\rangle_G = \frac{1}{m} \sum_{i=1}^{m} f(x_i) g(x_i) + \lambda_{Opticom} \langle \nabla f, \nabla g\rangle_2 \tag{2.34}$$

for each cell of the matrix.

The third approach uses the error of the component grids for finding suitable coefficients. For this approach we assign smaller coefficients to the component grids where the error is high. The c_i are then computed with

$$c_i = \frac{c_i}{error_i}. \tag{2.35}$$

Notice that this approach does not directly depend on the data set used. We also do not optimize in any way, i.e. the resulting error could possibly be worse than before.

As a last step of all three different approaches, the vector of the c_i has to be rescaled so that the sum equals one.

3 Implementation

The theoretical background explained so far is the foundation for the implementation. This chapter gives an overview of the usage and functionalities of the framework. Furthermore, the implementation of the regression using the Sparse Grid Combination Technique will be presented.

3.1 sparseSpACE framework

The sparseSpACE framework created by Michael Obersteiner offers many functionalities in the context of Sparse Grids. Especially when dealing with high dimensional data, the use of the framework leads to faster and more efficient solutions than using full grids. In the beginning, it was intended to integrate high dimensional functions. By now, many other functionalities were added, like *Density Estimation* by M. Fabry [9] and general *Machine Learning* tasks by C. Moser [10].

All the operations can be used in the context of the normal Sparse Grid Combination Technique or an adapted version that is spatially adaptive to the data points. Every operation is a subclass of `GridOperation` such as the existing class `DensityEstimation`. For more information about this class and the functionality, refer to [9].

To add the possibility of performing regression on an arbitrary data set, the class `Regression` as a subclass of `GridOperation` was added to the framework. Due to many similar functions that Regression and DensityEstimation share, these methods were moved to a more general class `MachineLearning` which is now the superclass of `DensityEstimation` and `Regression`.

In the following, the different ways of using the class will be introduced with information about how the methods are implemented. First, the general details about how the class stores data and how it can be used is explained in the beginning of section 3.2. Then the description of the operation will be split into two parts, the one using the normal Combination Technique (section 3.3) and the case of with the spatially adaptive version in section 3.4.

3.2 Regression class

There are different approaches and various parameters that can be selected when performing regression. These different possibilities can be set in the constructor of the class. The most important parameters are the following:

- `data`: list of vectors $\vec{x}$ of the data set
- `target_values`: list of target values y of the data set
- `regularization`: regularization parameter λ
- `regularization_matrix`: choice of the regularization matrix (I or C)
- `range`: specification of the range, to which the data will be scaled to

The data set of the constructor contains all data points of the different phases in machine learning. Therefore, the variables `training_data`, `training_target_values`, `validation_data`, `validation_target_values`, `test_data` and `test_target_values` store the set of each corresponding phase. In the constructor, the method `scale_data` is called which scales the data to the provided range. This is always a subrange of the hypercube as described in chapter 2.

The training data set is used for finding suitable weights for the basis functions of the grid during regression. For the Opticom, which is optional, the validation set is used. With these points, which also contain the training set, the weights of the single component grids are optimized which can be interpreted as hyperparameters. Finally, the error is measured with the test set. The mean squared error is used as error norm.

All in all, the following possibilities can be chosen to perform regression:

Normal Combination Technique	Spatially adaptive version
without regularization	without regularization
matrix I	matrix I
matrix C	matrix C
Opticom (3 different variants)	Opticom (3 different variants)

In the next subsections, the pipeline of the regression object is presented. First, the case for the normal Combination Technique is explained, which will be followed by the spatially adaptive methods. In both cases, the single steps are *training* the object, *optimizing* the coefficients with Opticom, and finally *computing* the test error.

3.3 Methods for normal Combination Technique

To provide a simple possibility to perform the single steps of the pipeline, the detailed steps are encapsulated in three functions for training, the Opticom and testing.

3.3.1 Training with the normal Combination Technique

After the initialization of the Regression object, the method

```
train(percentage_of_testdata, minimum_level, maximum_level, noisy_data)
```

can be invoked. More detailed, the function initializes the member variables storing the different data sets of the three phases (training, validation, and testing). The whole data set is partitioned as follows. We take `percentage_of_testdata`% as test data. The remaining part is used as validation set. Additionally, 85% of this set is used in the training phase. We also add white noise to the target values of the training and validation set if the boolean parameter `noisy_data` is set to true. The disturbance is dependent on the data. We first find the highest of all target values in terms of amount and then add gaussian noise with zero mean and 1% of the highest value as standard deviation. This increases the complexity of the regression task and allows to mimic real world problems that are often noisy. After this splitting and adaption of the target values, an object (`combiObject`) of class `StandardCombi` is initialized and the operation is performed on each component grid independently. That is where the weights of the basis functions are computed. The function returns the `combiObject` so that the invoker of `train` can also get further information about this object.

As mentioned in chapter 2, the regression is performed on every single component grid. This is done in the next step by the method `solve_regression(levelvec)` when performing without regularization term or `solve_regression_smooth(levelvec)` in case of $\lambda \neq 0$. The first variant can be seen in figure 3.1.

```
1 def solve_regression(levelvec):
2   A = self.build_A_matrix(levelvec)
3   y = self.training_target_values
4   alphas, res, rank, s = np.linalg.lstsq(A, y, rcond=None)
5   return alphas
```

Figure 3.1: Pseudo code to solve the regression without regularization

`solve_regression` refers to the simplest case which is described in equation 2.24. After calculating the matrix and the vector (3.1, lines 2 and 3), the alphas are found

with the help of the function `lstsq` from the library `np.linalg` (np short for numpy) [11]. This method solves the least squares problem and returns the solution called `alphas`, the residuals, rank of the matrix A and the singular values of A which are not needed in this case.

```
1 def solve_regression_smooth(levelvec):
2   left  = self.build_left_matrix(levelvec)
3   right = self.build_right_vector(levelvec)
4   alphas, res, rank, s = np.linalg.lstsq(left, right, rcond=None)
5   return alphas
```

Figure 3.2: Pseudo code to solve the regression with regularization

If the regularization parameter lambda is not zero, `solve_regression_smooth` is called (see figure 3.2). `build_left_matrix` and `build_right_vector` (3.2, lines 2 and 3) calculate the left and the right part of the linear system described in equation 2.29 or 2.30. This decision is made in the method `build_left_matrix` and depends on the chosen matrix when the regression object was initialized. The matrix C is built with the help of the method `build_C_matrix`. With the entries $(C)_{i,j} = \langle \nabla\varphi_i, \nabla\varphi_j \rangle_{L_2}$ which were described in section 2.3, the matrix is built as it can be seen in 3.3.

The method is based on the following idea. We can also write the entries of the matrix as

$$(C)_{i,j} = \langle \nabla\varphi_i, \nabla\varphi_j \rangle_{L_2} = \int_\Omega \nabla\varphi_i(\vec{x})^T \nabla\varphi_j(\vec{x}) d\vec{x} = \sum_{k=1}^{d} \int_\Omega \frac{\delta\varphi_i(\vec{x})}{\delta x_k} \frac{\delta\varphi_j(\vec{x})}{\delta x_k} d\vec{x} \tag{3.1}$$

with $\frac{\delta\varphi_i}{\delta x_k}$ being the partial derivative of the basis function of the k^{th} dimension. This can be further simplified to

$$\begin{gathered} \sum_{k=1}^{d} \int_\Omega \frac{\delta\varphi_i(\vec{x})}{\delta x_k} \frac{\delta\varphi_j(\vec{x})}{\delta x_k} d\vec{x} = \\ \sum_{k=1}^{d} \prod_{m=1}^{d} \int_{\Omega_m} \delta_{km} \varphi'_{i_m}(x_m) \varphi'_{j_m}(x_m) + (1 - \delta_{km}) \varphi_{i_m}(x_m) \varphi_{j_m}(x_m) dx_m \end{gathered} \tag{3.2}$$

with δ_{km} being the Kronecker delta and $\varphi_i(\vec{x}) = \prod_{m=1}^{d} \varphi_{i_m}(x_m)$. As a consequence we only have one-dimensional integrals over the 1d hat functions.

The two loops in lines 7 and 8 iterate over all cells $C_{i,j}$ of the matrix. Only one half of the matrix is calculated explicitly because of the symmetry of C. The third loop (line 10)

```
def build_C_matrix(levelvec):
    dim = len(levelvec)
    grid_size = self.grid.get_num_points()
    C = np.zeros((grid_size, grid_size))
    index_list = np.array(get_cross_product_range_list(grid.numPoints)) + 1
    # loop over all combinations of indices of basis functions
    for i in range(grid_size):
        for j in range(i, grid_size):
            res = 0.0
            for k in range(dim):
                temp_res = 1.
                for m in range(dim):
                    index_im = index_list[i][m]
                    index_jm = index_list[j][m]
                    if m == k:
                        # basis function overlap fully
                        if index_im == index_jm:
                            temp_res *= (2 ** (levelvec[k] + 1))
                        # basis function do not overlap
                        elif abs(index_jm - index_im) > 1:
                            temp_res = 0
                            break
                        # basis functions overlap partly
                        else:
                            temp_res *= -(2 ** (levelvec[k]))
                    else:
                        # basis function overlap fully
                        if index_im == index_jm:
                            temp_res *= 1 / (2 ** (levelvec[k] - 1) * 3)
                        # basis function do not overlap
                        elif abs(index_jm - index_im) > 1:
                            temp_res = 0
                            break
                        # basis functions overlap partly
                        else:
                            temp_res *= 1 / (2 ** (levelvec[k] - 1) * 12)
                res += temp_res
            C[i, j] = res
            C[j, i] = res
    return C
```

Figure 3.3: Pseudo code to build the matrix C

iterates over the dimensions to reference each level of the `levelvec` which stores the level of each dimension. With variable `m` and the fourth loop (line 12), each dimension of the basis functions is referenced. There are then two main cases. The first one where `m == k` (lines 16-25) and the second one from lines 27 to 36. In the former case, the derivative of the m^{th} dimension is multiplied for both basis functions. In the other case, the product of the two 1D basis functions in this dimension is multiplied. This idea is based on the fact that the basis functions are built with the tensor product which makes it possible to split the calculation in 1D opertions.

In both cases, there are three different possibilities. Either the basis functions overlap (completely or partly) or they do not. The different calculations deduce as follows. In line 18, the derivative of the product of fully overlapping basis functions has to be calculated. With the hat functions, this results in $(2^l)^2 \cdot \frac{1}{2^{l-1}} = 2^{2l} \cdot 2^{-l+1} = 2^{l+1}$ with l the level of the dimension, 2^l the gradient of the basis function and $\frac{1}{2^{l-1}}$ the width. An example for exactly overlapping basis functions can be seen in figure 3.4. There, the gradient reads $2^2 = 4$ and the width is $\frac{1}{2^{2-1}} = 0.5$.

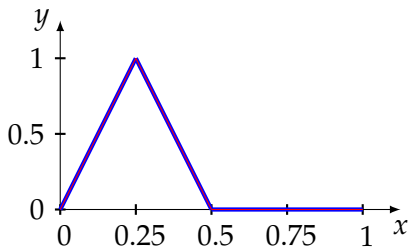


Figure 3.4: Examples for overlapping basis functions, in this example $l = 2$, gradient 4 and width 0.5.

In lines 20 and 21, φ_i and φ_j do not overlap which means that the product is zero. An example of two hat functions that do not overlap can be seen in figure 3.5. There, the product of the functions is zero leading to the gradient also being zero.

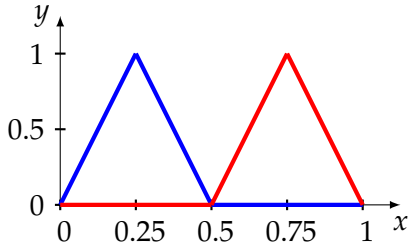


Figure 3.5: Examples for not overlapping basis functions, in this example the level is 2. The product of the basis functions is zero.

In line 25, the basis functions patially overlap. Therefore, one gradient is positive and one gradient is negative resulting in a product of gradients less than zero. The width is this time half as big as in the completely overlapping case. All in all this result in $-(2^l)^2 \cdot \frac{1}{2} \cdot \frac{1}{2^{l-1}} = -2^{2l} \cdot 2^{-1} \cdot 2^{-l+1} = -2^l$. An example for two partly overlapping hat functions can be seen in figure 3.6.

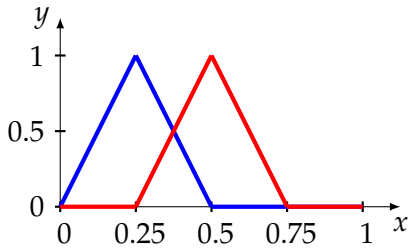


Figure 3.6: Examples for partly overlapping basis functions. In this example, the level is 2, the gradient -4 and the width of the overlapping interval is $\frac{1}{4}$

The cases from line 27 to 36 are from the right part of equation 3.2. Here, we directly multiply the hat functions without taking the derivative. The results can be summarized by

$$\int_{\Omega_m} \varphi_{i_m}(x_m)\varphi_{j_m}(x_m)dx_m = \begin{cases} \frac{1}{2^{l-1}\cdot 3} & \text{fully overlapping} \\ \frac{1}{2^{l-1}\cdot 12} & \text{partly overlapping} \\ 0 & \text{not overlapping} \end{cases}. \tag{3.3}$$

After calculating the values for each dimension, the results are added to the variable `res`. The corresponding cells are then assigned to this variable (lines 38, 39). Note that both $C_{i,j}$ and $C_{j,i}$ are assigned the same value because of the symmetry. This makes the method more performant.

3.3.2 Opticom with the normal Combination Technique

After the training phase, predictions can already be made. With the calculated alphas, data points can be interpolated with the `combiObject`. The next step in the pipeline is Opticom, where the hyperparameters can be optimized in the validation phase. Therefore, the method

```
optimize_coefficients(combiObject, option)
```

can be performed on the regression object. This function finds suitable coefficients of the component grids (*Opticom*). The option parameter determines, which approach is

used for Opticom. There are three possible values to choose from:

- 1: Least squares based approach with regularization term (described in equation 2.33)
- 2: Least squares based approach without regularization (based on the validation set, described in equation 2.32)
- 3: Error based approach (based on the errors of the component grids, described in equation 2.35)

For the first option, the matrix and vector described in 2.33 are built and the suitable coefficients for the component grids are then found with `lstsq` from numpy, a package with many functionalites for scientific computing in python. The entries of the matrix are computed as it can be seen in figure 3.7.

```
1  def build_matrix_opticom(combiObject):
2    matrix = np.zeros((len(combiObject.scheme), len(combiObject.scheme)))
3    vector = np.zeros((len(combiObject.scheme)))
4    # loop over all cells of the matrix
5    for i in range(len(combiObject.scheme)):
6      for j in range(i, len(matrix)):
7        # interpolate the validation set on component grid i and j
8        vec_i = self.interpolate_points_component_grid(combiObject.scheme[i], self.validation_data)
9        vec_j = self.interpolate_points_component_grid(combiObject.scheme[j], self.validation_data)
10       # sum the component wise product and scale
11       var_sum = np.dot(vec_i.flatten(), vec_j.flatten())
12       var_sum *= 1 / len(self.validation_data)
13       # add regularization term
14       var_sum += self.regularization_opticom * self.compute_regularization_term_opticom(
       combiObject.scheme[i], combiObject.scheme[j])
15       # fill matrix and vector
16       matrix[i][j] = var_sum
17       matrix[j][i] = var_sum
18       if i == j:
19         vector[i] = var_sum
20
21   return matrix, vector
```

Figure 3.7: Code to build the matrix and the vector of Opticom

In this method, the vector is computed at the same time as the matrix because the values from the diagonal are the same as the entries in the vector. Otherwise, this would be calculated more than one time. Again, the two loops (lines 5,6) iterate over all cells in the matrix. The two vectors `vec_i` and `vec_j` store the interpolated validation points. They are then combined with the scalar product and the factor $\frac{1}{m}$ with m being the size of the validation set. In the function `compute_regularization_term_opticom`, the same matrix C is built as in the regression, with the difference that the weights of the basis functions are multiplied with the single expressions. The method then sums up all entrties of the matrix and the result is then added in line 13. The regularization parameter is the same as the one used in the regression if it is not set explicitly. In most cases, we have change its value to obtain a reasonable result.

For the second option, also the matrix and the vector are built as described in equation 2.32. The cells of the matrix are simply filled with the validation points interpolated on the different component grids. In this case, also `lstsq` from numpy is used to solve this least squares problem. The coefficients are then updated and scaled so that their sum equals one.

The last approach that can be used for Opticom can be seen in figure 3.8.

```
1  def optimize_coefficients_error_per_grid(self, combiObject):
2      error_vec = np.zeros(len(combiObject.scheme))
3      coefficients  = np.zeros(len(combiObject.scheme))
4      # interpolate on each component grid and calculate the error
5      for i in range(len(combiObject.scheme)):
6          learned_targets = self.interpolate_points_component_grid(combiObject.scheme[i], validation_data)
7          error_vec[i] = sklearn.metrics.mean_squared_error(self.validation_target_values, learned_targets)
8          coefficients[i] = combiObject.scheme[i].coefficient
9      # update the coefficients of the grids depending on the error
10     for i in range(len(combiObject.scheme)):
11         coefficients[i] = coefficients[i] / error_vec[i]
12         # scale the coefficients so that the sum of the vector equals one
13     length = np.sum(coefficients)
14     for i in range(len(combiObject.scheme)):
15         combiObject.scheme[i].coefficient = coefficients[i] / length
```

Figure 3.8: Code to update the coefficients according to the errors per component grids

The first loop (lines 5 to 8) interpolates the validation set on each component grid and calculates the error with the actual target values. We use the mean squared error. It is calculated with a method from the `scikit-learn` library [12]. Some other functionalities from this library are used in other parts. In the second loop (lines 10, 11), the

coefficients are updated like it is described in 2.35. The other lines 13, 14 and 15 set the coefficients which are also scaled so that the sum equals one. This is necessary because after the update in line 11, this is not guaranteed anymore.

3.3.3 Testing with the normal Combination Technique

The last method in the pipeline which finally tests and evaluates the regression object is the function

```
test(combiObject).
```

The predicted target values $\tilde{y}_i$, which can be found using the combiObject and the actual y_i are evaluated using the mean squared error. It is calculated with

$$MSE = \frac{1}{m}\sum_{i=1}^{m}(y_i - \tilde{y}_i)^2. \tag{3.4}$$

These three functions make it easy to use the `Regression` class by encapsulating the methods that are used to reconstruct the function given by the data points.

3.4 Methods for the Spatially Adaptive Combination Technique

For this use case, similar functions are implemented. The pipeline consists of the same three functionalities. This time, the object that performs the operation has to be of type `SpatiallyAdaptiveBase` or a subclass of it. The class itself inherits from `StandardCombi` but implements more functionality which makes it possible to adapt the grid to the dataset.

3.4.1 Training with the Spatially Adaptive Combination Technique

The first method that has to be called is `train_spatially_adaptive()`. It has the parameters `percentage_of_testdata` which specifies how the data set is split, the `margin`, which is responsible for the adaptivity and the stopping criteria `tolerance` (for current error), and `max_evaluations` (refinement stops when current number of evaluations is greater or equal than the value of this parameter). The value of `do_plot` specifies whether the grid of each step of the refinement should be plotted and

`noisy_data` determines whether we add white noise to the target values of the training and validation set.

In the beginning of the function, the data set is split according to the specified sizes. The percentages for the training, validation, and test set are the same as in the case of the normal Combination Technique. In the following, variables that the `SpatiallyAdaptiveSingleDimensions2` object uses are instantiated. These are then, together with the second, third and fourth parameter, forwarded to this instance and they determine the behavior of the adaptivity. The grid refines in all regions where the error is greater than or equal to a percentage of the highest error. This number can be specified with `margin` which is always in the interval $[0, 1]$. The higher the margin the more fine-grained we refine. With a margin of 0 everything is refined equally. With `tolerance` and `max_evaluations`, the stopping criteria are set. If the whole error is smaller or equal than the tolerance or the number of evaluations exceeds the third parameter, the refinement stops and the weights of the basis functions are found.

After the initialization of the object with these parameters, the regression is performed. Again in this case, there are two major methods that perform the regression. The case without regularization is done by `solve_regression_dimension_wise`. It does essentially the same as the one performing regression on the normal Combination Technique. The matrix and the vector are built according to equation 2.24. The system is then solved using `lstsq` from numpy.

The other possibility is to use a regularization matrix I or C. This is the case when $\lambda \neq 0$. Then the method `solve_regression_dimension_wise_smooth` will be called. Again, the steps are the same as in the respective case with the normal Combination Technique. The difference is the calculation of the C matrix. This can be seen in the part of the method `build_C_matrix_dimension_wise` in figure 3.9.

In this method, the calculation of the integrals is a bit more complex because of the adaptivity. For example basis functions might have different widths in the adaptive case. And also the comparison of the indexes of hat functions is not sufficient because they might not overlap even though they have the same index. Another problem is that the hat function can also be asymmetric because the grid point is not exactly in the center of the interval anymore. An example can be seen in figure 3.10.

Although, the general structure is the same as with the normal combination technique. The whole block is splitted in the first part (where `m == k`) where the actual derivatives of the functions are computed and the second part where the normal hat functions are used. The calculation is again according to 3.2.

```
[...] if m == k: # derivatives of the basis functions are taken
        if no overlap:
          temp_res *= 0
        elif fully overlapping:
          b1 = point_i[k] - domain_i[k][0]
          m1 = 1 / b1
          b2 = domain_i[k][1] - point_i[k]
          m2 = 1 / b2
          temp_res *= b1 * m1 ** 2 + b2 * m2 ** 2
        else: # partially overlapping
          if point_i[k] < point_j[k]:
            b = point_j[k] - point_i[k]
            m = 1 / (point_j[k] - point_i[k])
            temp_res *= -(m * m * b)
          else:
            b = point_i[k] - point_j[k]
            m = 1 / (point_i[k] - point_j[k])
            temp_res *= -(m * m * b)
      else: # no derivatives of the basis functions are taken
        if kth coordinate of the points are not the same:
          m = 1.0 / abs(point_i[k] - point_j[k])
          a = min(point_i[k], point_j[k])  # lower end of integral
          b = max(point_i[k], point_j[k])  # upper end of integral
          integral_calc = lambda x, m, p, q: 0.5 * (m ** 2) * (x ** 2) * (p + q) - (1 / 3) * (
          m ** 2) * (x ** 3) - x * (m * p + 1) * (m * q - 1)
          integral = integral_calc(b, m, a, b) - integral_calc(a, m, a, b)
          temp_res *= integral
        else: # kth coordinate are the same
          if point_i[k] != domain_i[k][0]: #kth coordinate of p_i is not the left boundary
            m1 = 1.0 / abs(point_i[k] - domain_i[k][0]) # left slope
            integral_1 = lambda x, m, p: -((m * (p - x) - 1) ** 3 / (3 * m))
          else: #kth coordinate of p_i is the left boundary
            integral_1 = lambda x, m, p: 0
            m1 = 0
          if point_i[k] != domain_j[k][1]: #kth coordinate of p_j is not the right boundary
            m2 = 1.0 / abs(domain_j[k][1] - point_j[k])  # right slope
            integral_2 = lambda x, m, p: -((m * (p - x) + 1) ** 3 / (3 * m))
          else: #kth coordinate of p_j is the right boundary
            integral_2 = lambda x, m, p: 0
            m2 = 0
          a = domain_i[k][0] # lower end of first integral
          p = point_i[k]  # upper end of first integral, lower end of second integral
          c = domain_i[k][1] # upper end of second integral
          integral = (integral_1(p, m1, p) - integral_1(a, m1, p)) + (
          integral_2(c, m2, p) - integral_2(p, m2, p))
    temp_res *= integral [...]
```

Figure 3.9: Code snippet to calculate the C matrix for the spatially adaptive use case

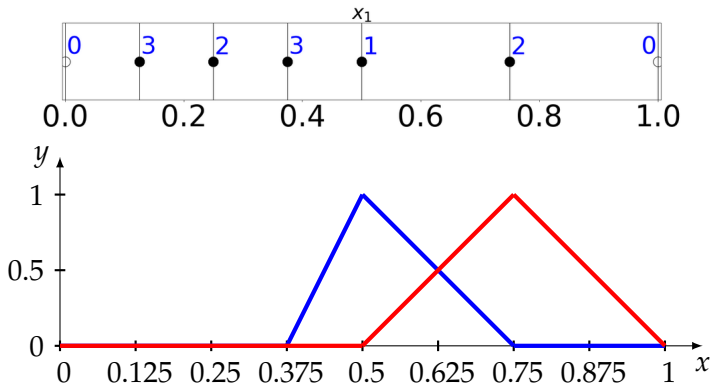


Figure 3.10: An example for an adaptive grid in one dimension is above. The corresponding hat function can be seen right below.

3.4.2 Opticom with the Spatially Adaptive Combination Technique

The next step after training the regression object in the pipeline is to optimize the coefficients of the component grids with Opticom. This can again be done with three options. The different possibilities are the same as in the case with the normal Combination Technique. Only the implementations differ slightly. The optimization can be started with the method `optimize_coefficients_spatially_adaptive( adaptiveCombiInstance, option)`. The numbers $1-3$ for option are the same as for `optimize_coefficients(...)`.

For the first approach which is the one described in equation 2.33, the matrix and the vector are built. The overall system is then solved with `lstsq` from numpy. The part that is different is the projection of the basis functions on the same grid. Remember that for the entries of the matrix in 2.33 possibly a higher level has to be used. This is implemented in the function `compute_regularization_term_opticom_spatially_adaptive` which can be seen in figure 3.11.

In lines 6 and 7, the maximum level in each dimension is calculated. The adaptiveCombiSingleDim object stores the information, how the grid is refined in which levels. The list of points per dimension can then be retrieved with the method `get_point_coord_for_each_dim(levelvec_new)` (line 10). With that information, all grid points can be calculated in lines 11-13. Therefore, the weights of all basis functions can be calculated which are important for the sum of the matrix. As a result, the function returns the sum of the matrix which is the regularization term for this opticom method.

As second option for opticom (see also equation 2.32), the method

```
def compute_regularization_term_opticom_spatially_adaptive(adaptiveCombiInstanceSingleDim,
    component_grid_i, component_grid_j):
  levelvec_i  = component_grid_i.levelvector
  levelvec_j  = component_grid_j.levelvector
  levelvec_new = np.array((levelvec_i))
  # calculate the highest level in each dimension
  for i in range(len(levelvec_i)):
    levelvec_new[i] = max(levelvec_j[i], levelvec_i[i])
  # calculate the concrete grid points depending on the levels
  points_per_dimensions_list = []
  point_coords, point_levels, children_indices = adaptiveCombiInstanceSingleDim.
    get_point_coord_for_each_dim(levelvec_new)
  for d in range(len(levelvec_new)):
    points_per_dimensions_list.append(point_coords[d])
  evaluation_points = get_cross_product_list(points_per_dimensions_list)
  # calculate the weights of the basis functions (needed for matrix C)
  alphas_i = adaptiveCombiInstanceSingleDim.interpolate_points(evaluation_points, scheme_i)
  alphas_j = adaptiveCombiInstanceSingleDim.interpolate_points(evaluation_points, scheme_j)
  sum_all = self.sum_C_matrix_with_alphas_spatially_adaptive(point_coords, alphas_i, alphas_j)
  return sum_all
```

Figure 3.11: Code to calculate the regularization term for Opticom in the spatially adaptive case

`optimize_coefficients_minimize_whole_error_spatially_adaptive` adapts the coefficients of the component grids with the help of the validation set. The steps are the same as in the version without the spatial adaptivity. Only the interpolation of the points is slightly different.

Also the third option is similar to the version without spatially adaptivity. In this case it also holds that lower errors in a grid result in higher coefficients of this component grid. This option is used when the value of the second parameter is is set to 3.

3.4.3 Testing with the Spatially Adaptive Combination Technique

The last step in the pipeline of the spatially adaptive Combination Technique is testing. The method `test_spatially_adaptive(adaptiveCombiInstanceSingleDim)` has to be called therefore. As function parameter, an object of type SpatiallyAdaptivBase has to be passed. With this, the interpolation of the validation set is possible. The resulting predictions of these points are then again compared with the actual target values. The same norm as in the case with the normal Combination Technique is used (see also 3.4).

4 Results

The implementation of the different regression options is evaluated in this chapter. Therefore, different configurations are tested regarding how well the regression performs. One big advantage of Sparse Grids is the reduction of the complexity while still producing sufficiently good results as already mentioned in chapter 2. This property will be tested in the beginning followed by different configurations and options of the regression. We will use various functions and datasets to evaluate certain properties of the implementation. In some cases, we add white noise to the target values of the training and validation set. The test functions will be the following:

$$f_{gauss}(\vec{x}) = e^{-\sum_{i=1}^{d} k_i \cdot |x_i - p_i|}$$

$$f_{discont}(\vec{x}) = \begin{cases} 0 & \text{, if } \vec{x} \geq \vec{p} \\ e^{-\sum_{i=1}^{d} k_i \cdot x_i} & \text{, otherwise} \end{cases}$$

$$f_{expvar}(\vec{x}) = (1 + \frac{1}{d})^d \cdot (\prod_{i=1}^{d} x_i^{\frac{1}{d}})$$

$$f_{poly}(\vec{x}) = \prod_{i=1}^{d} c_i \cdot x_i^a$$

$$f_{oszillatory}(\vec{x}) = \cos(2\pi \cdot b + \sum_{i=1}^{d} c_i * x_i)$$

$$f_{C0}(\vec{x}) = -\sum_{i=1}^{d} c_i \cdot |x_i - p_i|$$

With the parameters $\vec{k}$ and $\vec{p}$, the width and location of the function maximum can be specified for the first one. The vector $\vec{p}$ in the second (discontinuous) function determines where the function is not continuous. With the $\vec{k}$, the gradient can be adapted. The third function (expvar) does not have additional parameters. Only the dimension can be changed. For the polynomial function, the degree a and the coefficients $\vec{c}$ in each dimension can be specified. The b in the oszillatory function describes the offset, whereas the $\vec{c}$ specifies the frequency of each dimension. The last

function (C0 or star function) can be adapted by changing the maximum point ($\vec{p}$) and the coefficients ($\vec{c}$).

Additionally, the following three data sets which are available in the *scikit-learn* library are used. The dimensionality of the data points is denoted with d.

Name of data set	d	Number of samples	Interval (target values)
Boston house-prices	13	506	$[5, 50]$
Diabetes	10	442	$[25, 346]$
California housing	8	20640	$[0.15, 5]$

Table 4.1: Data sets available at the library *scikit-learn*

4.1 Regression with the normal Combination Technique

4.1.1 Full vs. Sparse Grids

The first aspect that will be evaluated is how the regression with the Sparse Grid Combination Technique performs in comparison with full grids. Therefore, two functions are used to compare the results of the regression. The Sparse Grid is built with the Combination Technique with minimum level 1 and maximum level of 3. The full grid is also built with the Combination Technique but with both levels 3. The resulting grids in two dimensions can be seen in figure 4.1.

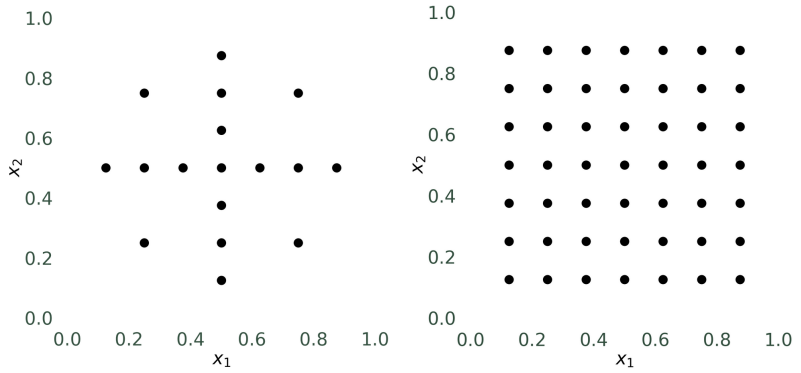


Figure 4.1: On the left, the Sparse Grid can be seen. It is the result of the Combination Technique with minimum level 1 and maximum level 3. On the right, the full grid with level 3 is depicted. The number of grid points is much higher than left.

The focus of this comparison will lay on the impact of the dimension on the error of the regression and the time used for this computation. Two functions are used for this evaluation. The dimensions range from 2 to 5 and for each data set, the error of the predictions and the time is measured. The functions do not have noisy data and for each grid, no regularization term (regularization parameter $\lambda = 0$) is used. The size of the data set is 200 in each case. The results of the predictions using the oszillatory function (parameters $(5, 5)$ and 0) and the polynomial function (degree 2, and coefficients 2 in each dimension) can be seen in figure 4.2 in the left column. For both functions, the plot is depicted on the left in two dimensions, the error is shown in the center and the comparison of the time is on the right.

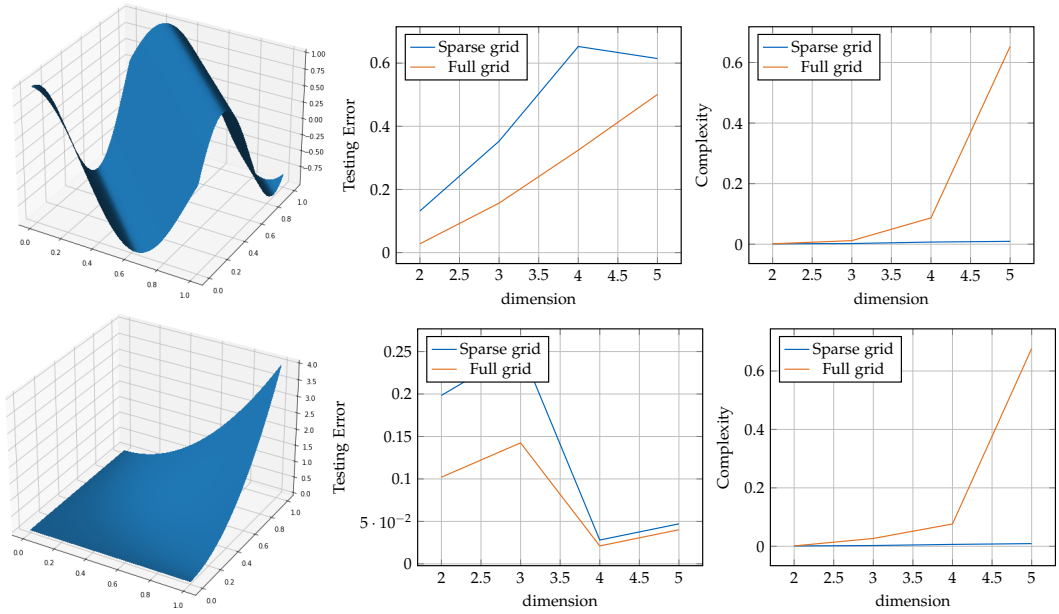


Figure 4.2: Error of predictions and time used with respect to different dimensions for full and Sparse Grids. Oszillating function on top and the polynomial function in the bottom.

For both functions, the results are similar. With increasing dimension, the complexity of the regression on full grids scales much higher than in the case with Sparse Grids. Already at dimension 5, the time is approximately $65 - 70$ times higher than using Sparse Grids. Although the times of both grids differ so much, the accuracies do not. Notice that the times of the Sparse Grids are not zero. This plot only depicts the high difference between those two fundamental grid types.

In figure 4.3, the combination scheme can be seen for the first function in the second

dimension. The result of the whole scheme is at the bottom.

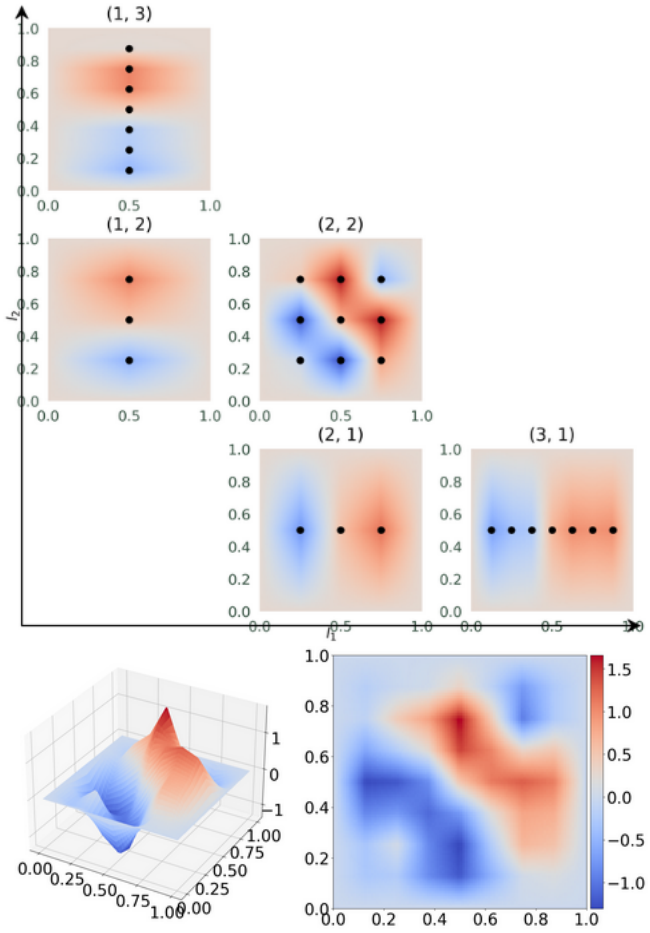


Figure 4.3: Regression on each component grid of the combination scheme for the oszillating function (top). Combined result in the bottom.

The reason for the big differences of the times in higher dimension is the *curse of the dimensionality*. In the one dimensional case, the full grid has $2^3 - 1 = 7$ grid points. This scales exponentially with the dimension leading to much slower computation times than using Sparse Grids. In general, the dimension of the data sets for the regression tasks will be much higher, but this is not feasible for full grids. This is why the upper bound of the dimension in this example is 5.

What is especially interesting in this comparison of full and Sparse Grids is the direct

dependency of the accuracy and the time. In figure 4.4, the time used to reconstruct the ExpVar function is plotted depending on the testing error. We compare how this develops in different dimensions (2, 3, and 5). In each case, 1000 data samples without additional noise are used. In the first two dimensions, the minimum level $l_{min} = 1$ and l_{max} is variable reaching from 2 to 5 for the Sparse Grids. The full grid has also the levels two, three, four, and five. We only use the levels two and three for the five dimensional case because of memory limitations.

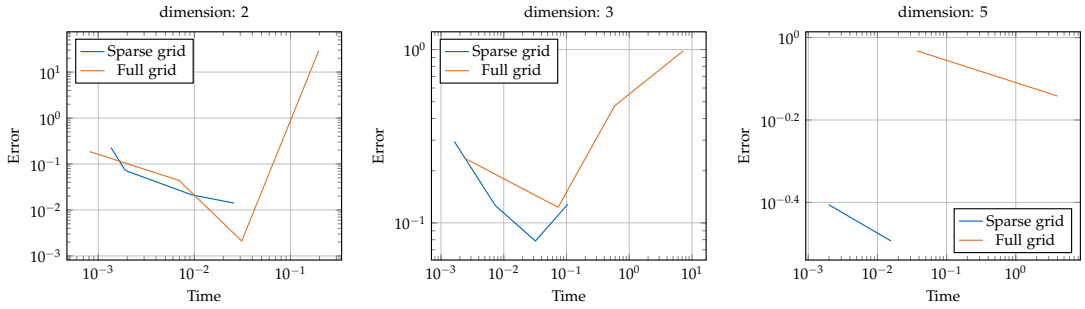


Figure 4.4: Time-error-plot for regression of the polynomial function in three dimensions with 1000 data points, full grids and Sparse Grids are compared in dimension 2 (left), dimension 3 (center) and dimension 5 (right).

In this diagram, an ideal method would be located in the bottom left corner. In general, a curve laying more bottom left than another one is dominating another one laying more in the top right corner. In this comparison, we can see a clear trend. The higher the dimension is, the more dominating is the curve of the Sparse Grid. While in the two dimensional case, only parts of the blue curve are dominating the orange one, the advantages of the Sparse Grids get clearer in the three and five dimensional case. Notice that in lower dimensions, full grids can possibly dominate Sparse Grids. We can observe that in the center, the curves are still located next to each other whereas in the right case with dimension 5, the Sparse Grids curve clearly dominates the orange one from full grids. The reason therefore is again the curse of the dimensionality which is tackled by the reduced number of grid points in the Sparse Grid case. We can obviously follow that using Sparse Grids makes much more sense in high dimensions.

4.1.2 Regularization parameter and term

The next criterion that has to be evaluated is the comparison of the two regularization terms. These are implemented as matrices I and C and are described in 2.29 and 2.30. For this comparison, the regularization parameter is variable and increasing. Again, the interesting results are the error of the predictions and the time used for the operation.

As already mentioned in chapter 2, the regularization parameter depends on the data set used. Two functions are evaluated to investigate the advantages and caveats of both matrices and also the optimal λ for each data set. The samples are in the first case from the discontinuous function (parameters $(4, 4)$ and $(0.9, 0.9)$, only a small difference of the target points in the discontinuous region) and in the second one from the gaussian function (maximum at $(0.5, 0.5)$ and parameters $(5, 5)$). For both tests, the number of data points is 1000 and a grid with minimum level 1 and maximum level 5 is used. The percentage of test data is 20% and no noise is added to the target values. The results of the tests can be seen in figure 4.5.

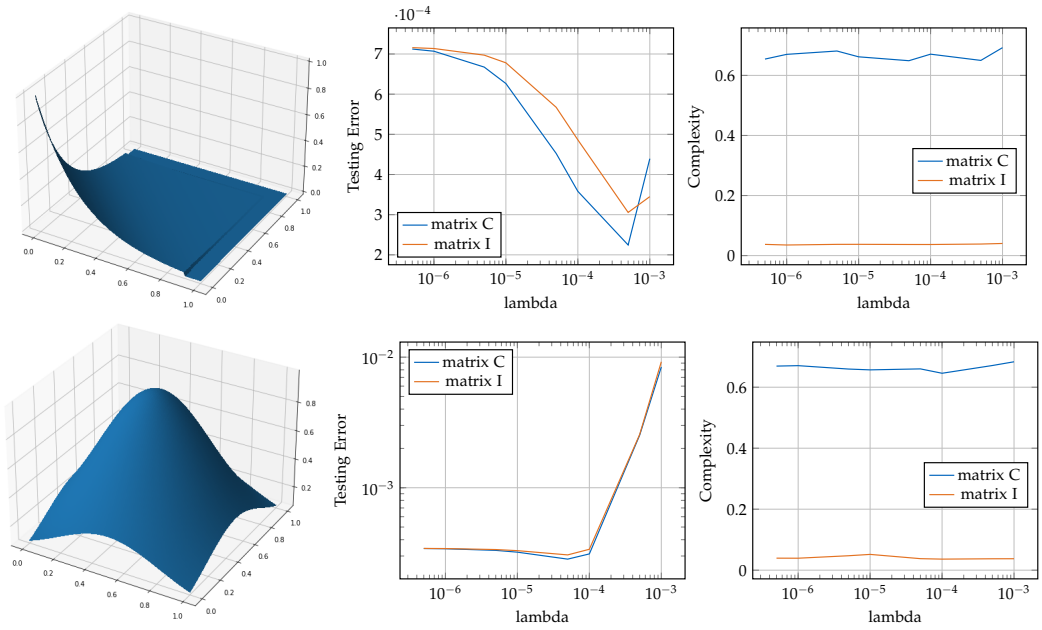


Figure 4.5: Plots of the error and time with respect to the regularization parameter λ and the function used as data set on the left. Functions discontinuous (top row) and gaussian (bottom row) used.

From these tests, we can observe different properties for the matrices C and I, but also for the regularization term and the parameter. The first one is that the error with matrix C and a suitable λ is slightly smaller than when using the matrix I, as expected. This is due to the fact that the matrix is better adapted to the basis functions when using this matrix. Although, regression with matrix I also performs very well (no significant difference to the error with matrix C).

Another finding from these tests is that the complexity of matrix C is very high. With $\mathcal{O}(n^2) \times \mathcal{O}(d^2)$ with n being the number of grid points and d being the dimension, this

especially increases with higher levels. Although the time used is much higher than for matrix I, the accuracy is not much better. That is a big disadvantage of C and an argument to choose I for regression tasks in high dimensions with many grid points where the use of C lead to too high computation times.

The last observation from this test is that the parameter λ really depends on the data set. In this case, the best choice is $\lambda = 5 * 10^{-4}$ for the first data set and $\lambda = 5 * 10^{-5}$ for the gaussian function. An example for the impact of λ on the result can be seen in figure 4.6. The left plot is the same function from above, the graph in the center is the plot of the solution with the best parameter and the right one is the result of a λ that is not suitable for this data set.

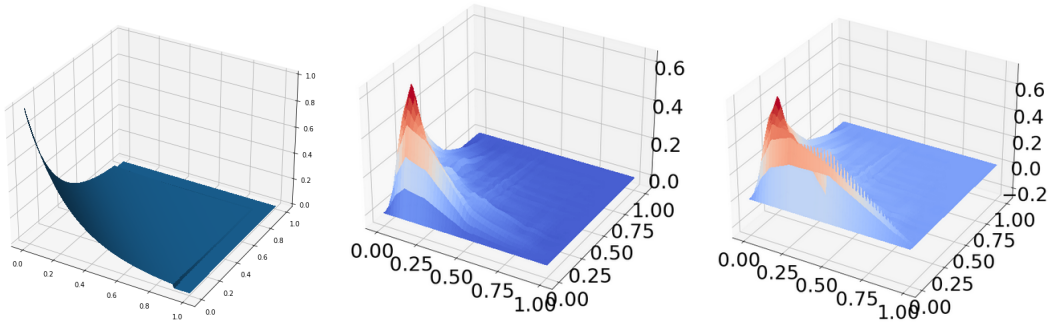


Figure 4.6: Example plots for different regularization parameters. On the left the function that has to be reconstructed (discontinuous). In the center the result for $\lambda = 0.0005$ and on the right with $\lambda = 10^{-5}$.

4.1.3 Different Opticom options

With the regularization parameter λ, there already comes a possibility to improve the result of the regression in the training phase. The step in the pipeline that was left out so far is to adapt the coefficients of the component grids. This optimization of the combination scheme was not made so far. In this section, the three different approaches for Opticom that are explained in chapter 2 are compared with each other and the normal regression. The error of the test set as well as the complexity will be the focus of the following.

The first measurement compares the three Opticom variants using a small example. The dataset used consists of samples of the oszillatory function in two dimensions and coefficients $(20, 5)$ with white noise added to the target values of the training and validation set. The number of points is 200. The regression is performed on a grid

resulting from the Combination Technique with $l_{min} = 1$ and $l_{max} = 3$. For the first Opticom approach, two test runs are made. We compare the method once without optimizing its $\lambda_{Opticom}$ and the second time with an optimized parameter $\lambda_{Opticom}$. The matrix used as regularization term is I. The results can be seen in table 4.2. We only compare the times of Opticom.

Opticom variant	error (MSE)	time of Opticom (s)
Without	0.49425	0.0
Garcke optimized	0.42505	0.26471
Garcke without optimization	0.46344	0.28221
Least squares based without regularization	0.47546	0.05977
Error based	0.47648	0.00213

Table 4.2: Comparison (with function Oszillatory) of the different Opticom versions with regression with the normal combination technique (without additional Opticom). Error and time are measured.

The same experiments are also made with the polynomial function (coefficients 0.8 in each dimension). The results are depicted in table 4.3.

Opticom variant	error (MSE)	time of Opticom (s)
Without	0.00369	0.0
Garcke Optimized	0.00354	0.28006
Garcke without optimization	0.00355	0.26735
Least squares based without regularization	0.00357	0.14903
Error based	0.00426	0.00206

Table 4.3: Comparison (with polynomial function) of the different Opticom versions with regression with the normal combination technique (without additional Opticom). Error and time are measured.

The most important findings from this experiment are that the regularization parameter for Opticom has to be adapted for good predictions and that the overhead that the variant introduced by Garcke [8] brings is very high. Without the additional optimization of $\lambda_{Opticom}$, the errors of all three variants can be compared. But after the adaption, the predictions are much better. It can be observed that the complexity of the first method (Garcke) is especially for high dimensional problems too high because of the regularization term of each entry in the matrix described in 2.33. The computation of the matrix and the vector is in $\mathcal{O}(|grids|^2 \cdot |grids| \cdot |grid\_points|^2 \cdot dim^2)$ with dim

being the dimension. That is the reason why further experiments will be made with the least squares based opticom ansatz without regularization.

To also evaluate the accuracy of regression with the optimization of the coefficients of the component grids in higher dimensions and with more data points, the two data sets diabetes and boston housing from `sklearn` were used. This time, the focus will be on the evaluation of Opticom as well. Additionally, the impact of the regularization will also be assessed. Two tests are made with the difference that one uses regularization and the other one does not. Each time and for each dataset, the regression is performed once with the Opticom and the other time without it. Additionally to the two datasets from the library, artificial function datasets in dimension 7 are used. The C0 function (star function) has parameters $(2, 2, 2, 2, 2, 2, 2)$ and the maximum is located at the exact center. The coefficients for the oszillatory function are $(20, 20, 20, 20, 20, 20, 20)$ and 0. For both functions, 400 samples are taken. In all experiments, the training and validation sets are disturbed with gaussian noise. The results are shown in figure 4.7.

For all data sets, the results look slightly different. This is because of certain properties of the data sets and the Opticom. For the first set in the top left corner, the Opticom optimizes the coefficients very well. In both cases, the red bar is much smaller than the blue one. What you can also see is that the error of predictions with regularization term is in general smaller than the ones where $\lambda = 0$. In this case, performing the Opticom brings a big advantage.

For the second dataset (boston, top right corner) this is not always the case. The accuracy of the regression with regularization decreases when performing opticom. The red bar is slightly higher than the blue one for the case with $\lambda \neq 0$ and matrix I with one exception in the top left diagram. The reason is that the regularization parameter already improves the results very much so that an optimization of the combination scheme does not optimize the results and also the grid level is suitable for a very accurate result. This is indicated by the very high blue bar without regularization and the small blue bar in the case with I.

All in all the two data sets in the top row have much higher errors than the other two datasets. This is due to the fact that the target values of them have higher values while the ones from the data sets at the bottom row are in the interval $[-1, 1]$. This generally leads to higher deviations.

The two data sets at the bottom have similar behavior. For both, the Opticom improves the accuracy in every case, and also the regularization leads to smaller errors.

In general, the Opticom is very useful to improve the coefficients of the component grids leading to higher accuracies in predicting new data points. Only few exceptional cases exist where no improvement is made with the Opticom.

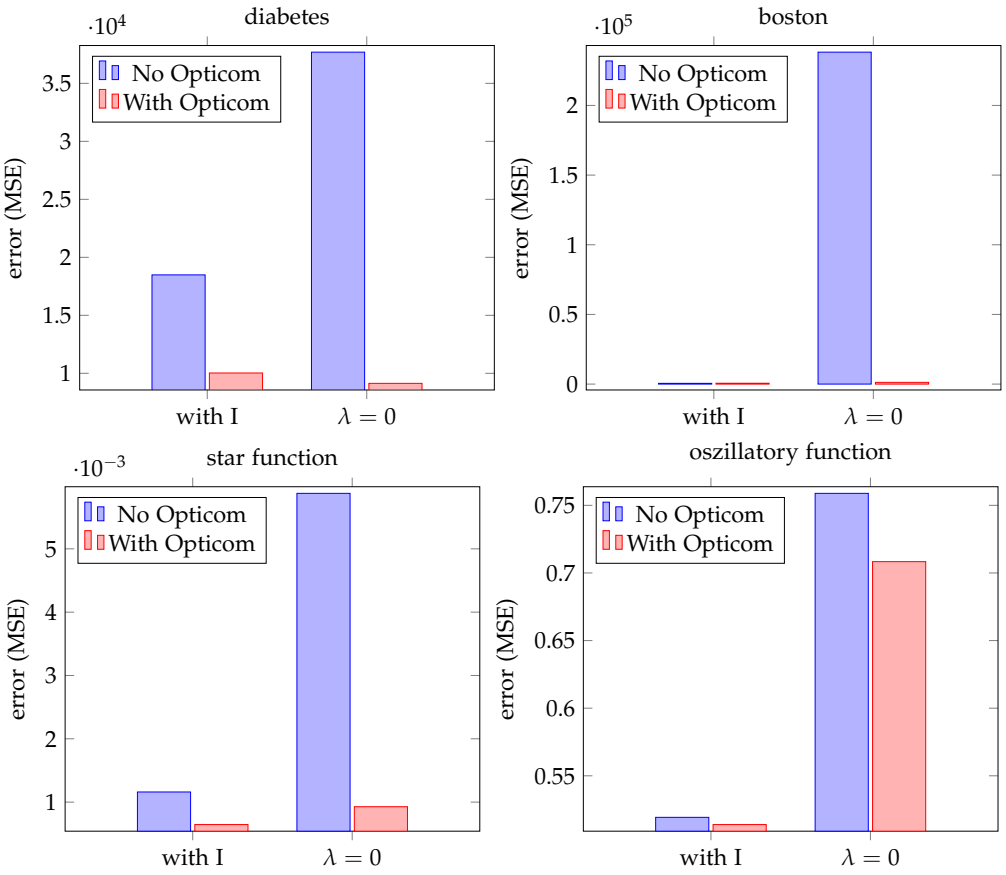


Figure 4.7: Errors of regression with/ without regularization, each with or without Opticom plotted for four data sets.

4.2 Regression with spatially adaptive Combination Technique

4.2.1 Margin and number of grid points

While the previous experiments showed results for the regression with the normal Combination Technique, the following ones use the spatially adaptive version. As already mentioned in chapter 2, the grids now adapt to the data set that is being processed. This happens through iteratively refining the grid in the regions where the error is still high. One important parameter in this setting is the `margin`. With this value from the interval $[0, 1]$, it can be specified where the grid has to refine in each step. At each grid point, the current error is computed. Now all points where this error is greater or equal than $margin \cdot error_{max}$ are being refined. This means that for $margin = 0$, a normal Combination Technique is used and the higher this value is, the more localized the refinement. With the first experiment, this impact of the parameter margin on the error is evaluated. It will also be important for further experiments.

As data set, random samples of a gaussian function with maximum at the point $(0.8, 0.8)$ are used. The plot of the function and the samples can be seen in figure 4.8.

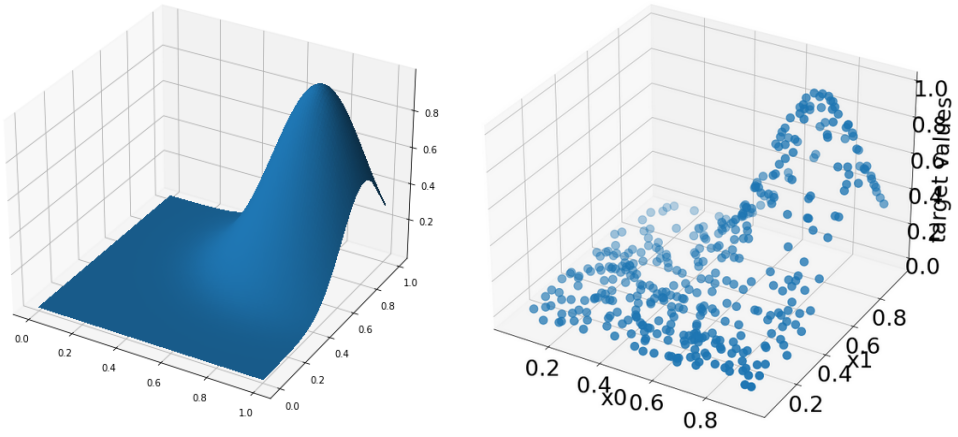


Figure 4.8: The gaussian function with maximum at $(0.8, 0.8)$ and coefficients 15 in each dimension plotted (left) and the samples of the function on the right.

The 400 data samples of the function are used with additional white noise. As stopping criteria, the maximal number of evaluations is set to 90 and the tolerance is chosen to be 10^{-3}. 20% of the data set is used for testing. For each case, a suitable regularization parameter λ is computed and used. In figure 4.9, the impact of the margin on the error can be seen.

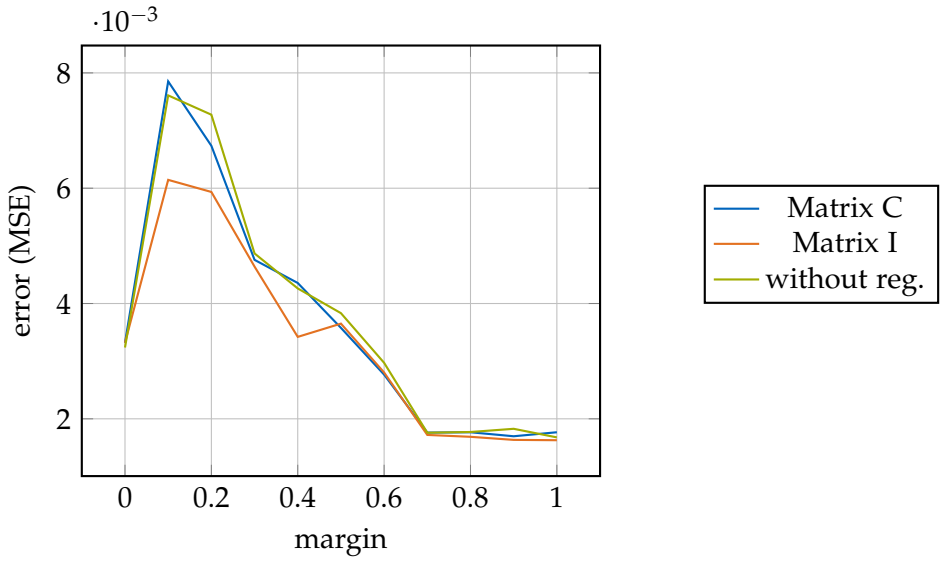


Figure 4.9: Error of the regression (MSE) depending on the margin. Comparison of regression with regularization (I and C) and without. Gaussian function was used.

The best value for the parameter margin is between 0.7 and 1. If it is chosen too small, then the grid is not adaptive enough to produce suitable results and the grid is too similar to the normal Sparse Grid. The special case is when $margin = 0$. Then the normal Combination Technique is used. An example for the spatially adaptive combination scheme with the resulting Sparse Grid and the solution of the regression can be seen in figure 4.10. Notice that no exact number of evaluations is taken into account in this experiment. It might be the case here that the final grid of the different cases with the various margins have not the same number of grid points. This might be lead to misleading interpretations regarding the amount of function evaluations. The refinement stops as soon as the current number of grid points is higher or equal than the given parameter which is in this case 90.

In this example, the errors first increase with increasing margin. However, then the errors constantly decrease until they reach a value of about half the error of the normal Combination Technique. This shows that an adaptive scheme can outperform the standard combination with margin $= 0$.

This can not be generalized because for most data sets, a value of 1 for margin does not lead to the highest accuracy. This can be observed in the next example. The data set used is based on the gaussian function again but this time with maximum at $(0.7, 0.7)$ and higher coefficients in each dimension leading to a higher width. The plot of the

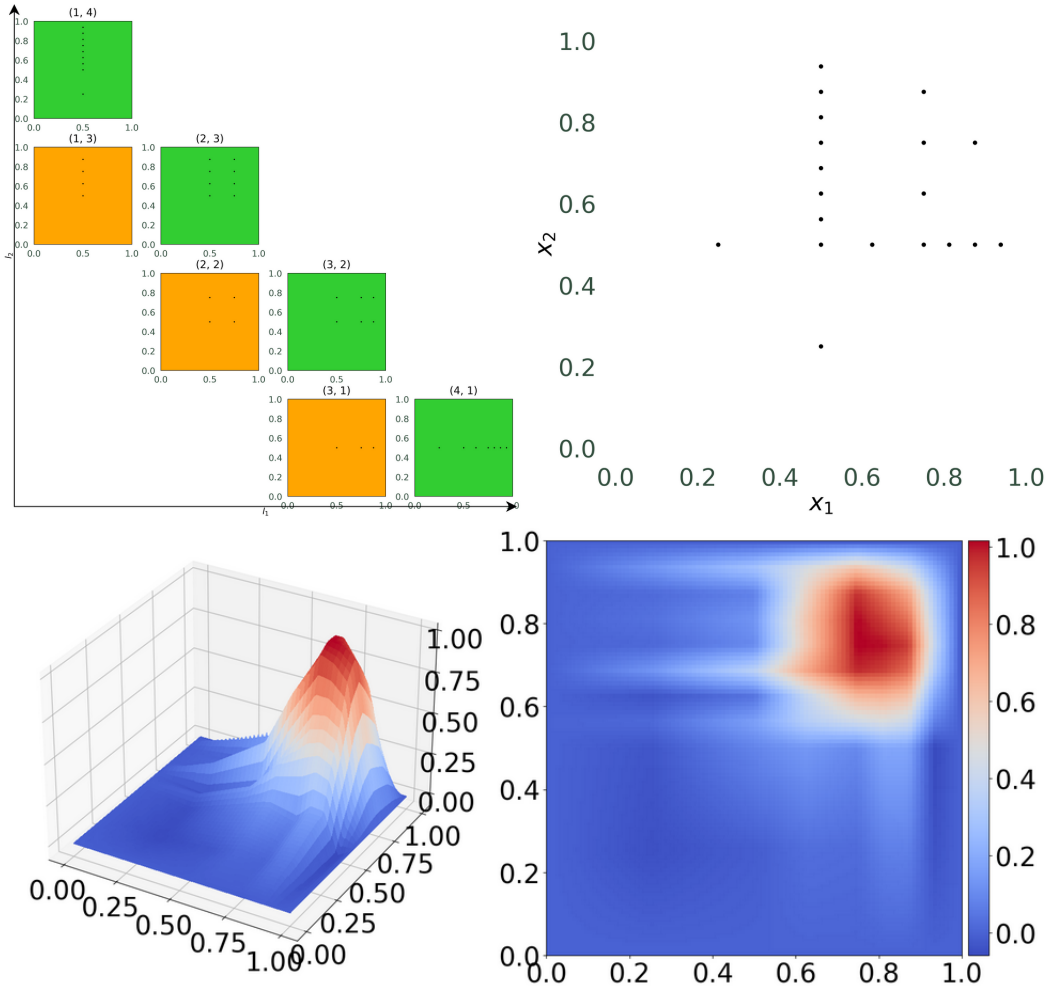


Figure 4.10: Example for spatially adaptive combination scheme with the gaussian function as data set. Regression was executed with regularization matrix I and $margin = 0.7$.

function and the samples used can be seen in figure 4.11.

The results of the regression with the same parameters as the previous gaussian example can be seen in figure 4.12.

In this example where the maximum is more centered, the error of regression does not decrease again for all regularization versions. For matrix I, the accuracy even gets worse for margin set to 1. Again notice that the exact number of grid points is not considered here. In general, this example is not as suitable for the adaptivity as the

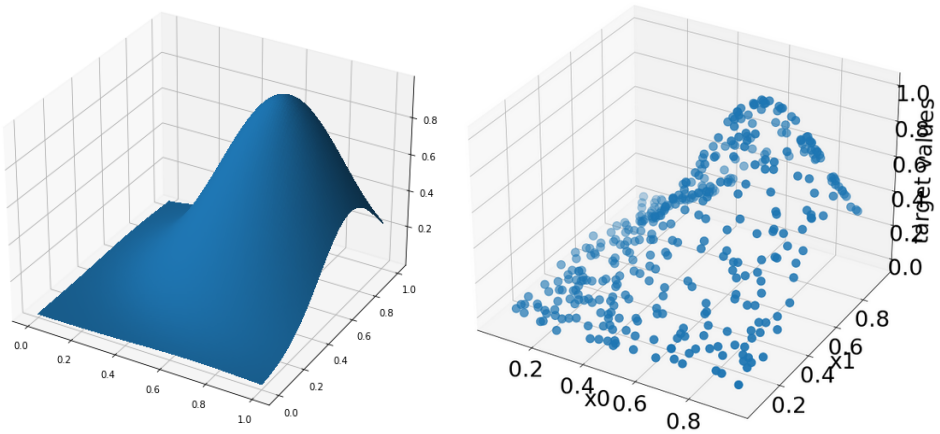


Figure 4.11: The gaussian function with maximum at $(0.7, 0.7)$ and parameter $(8, 8)$ plotted (left) and the samples of the function on the right.

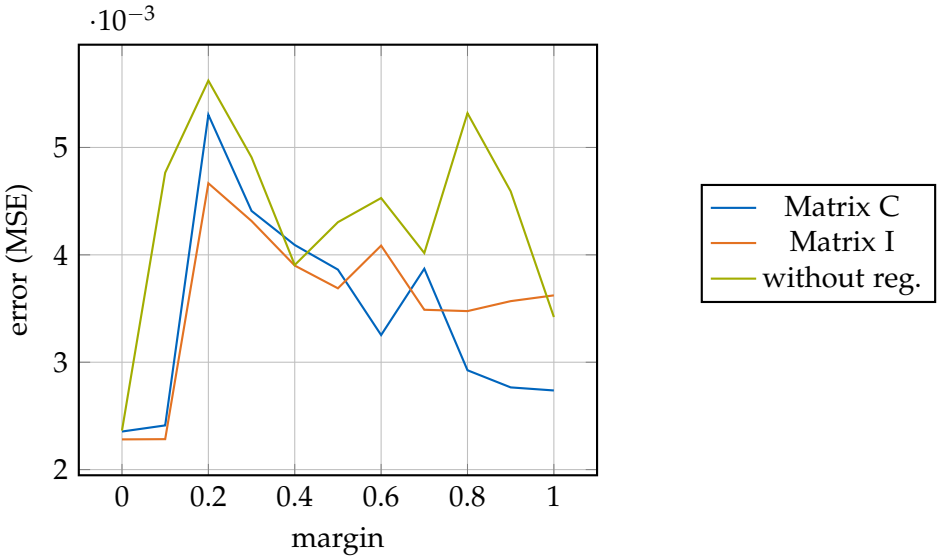


Figure 4.12: Error of the regression (MSE) compared to margin. This time, gaussian function with higher parameters in each dimension was used.

other gaussian function because of the slower descent and the more centered location of the maximum. That is also the reason why the normal Combination Technique brings the best results in this example. Another reason could be that the adaptive Combination Technique already leads to an overfitted solution. We focus on this aspect

in the following experiment. However, it seems to be consistent that it is sufficient to use the cheaper regularization matrix I in most cases.

All in all the parameter *margin* has big impact on the results of the regression and a suitable value has to be chosen to obtain good predictions. By now, no concrete number of refinement steps or grid points were considered. This is now done in the following test. It compares six different values for margin how the error of the regression develops after each refinement step. Figure 4.13 depicts the results. The gaussian function is used with dimension 5, and a maximum at $(0.8, 0.8, 0.8, 0.8, 0.8)$. The parameter is 20 in each dimension and 1000 samples are used.

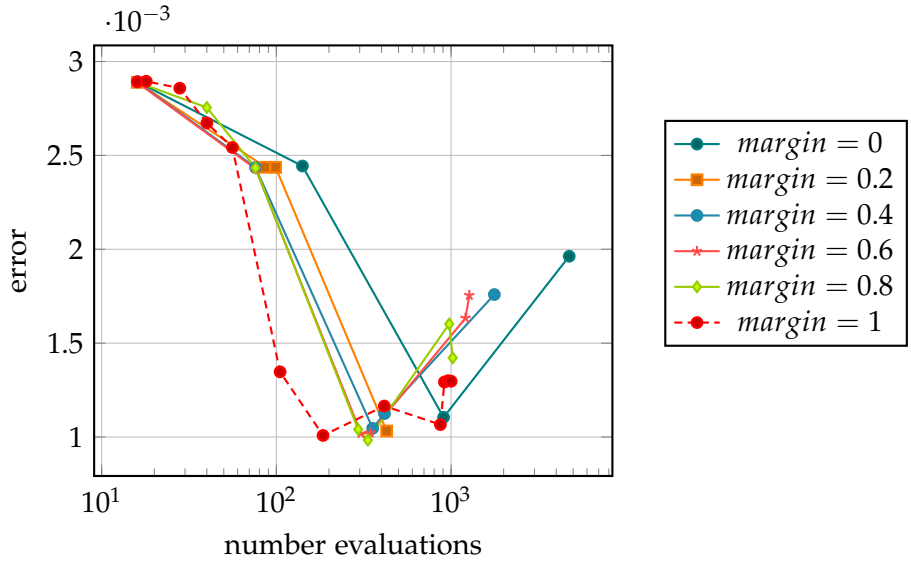


Figure 4.13: Testing error depending on the number of evaluations for different values of margin.

All curves first go down with increasing number of evaluations. This is where the best solution is found. After this minimum, the error increases again because an overfitted solution is found. This is caused by the too high number of grid points of the grid. The higher the margin is, the smaller is the minimum error. The reason therefore is that the grid adapts best to the given training points. Compared to the smaller values for margin, the minimum is found with a smaller number of evaluations because no unnecessary grid points are added. In general, a higher value for margin means a minimum error with fewer evaluations. But the accuracy decreases faster after this point again because the grid finds a highly overfitted solution faster than with smaller values for margin.

The impact of the number of grid points on the results of the regression can already

be seen in figure 4.13. A more detailed focus is laid on the maximal evaluations in the following experiment. To evaluate this behavior, the error of the regression using the spatially adaptive Combination Technique is measured for three variants of the gaussian distribution. The plots can be seen in figure 4.14. The maximum is located at $(0.7, 0.7)$ for all three data sets and the parameters are 10, 20, and 30 respectively. For all tests, 600 samples are taken without any additional noise. The margin is set to 0.7 and 20% of the samples are used as test set. Different values are taken for the `max_evaluations` parameter as it can be seen in the results from figure 4.15.

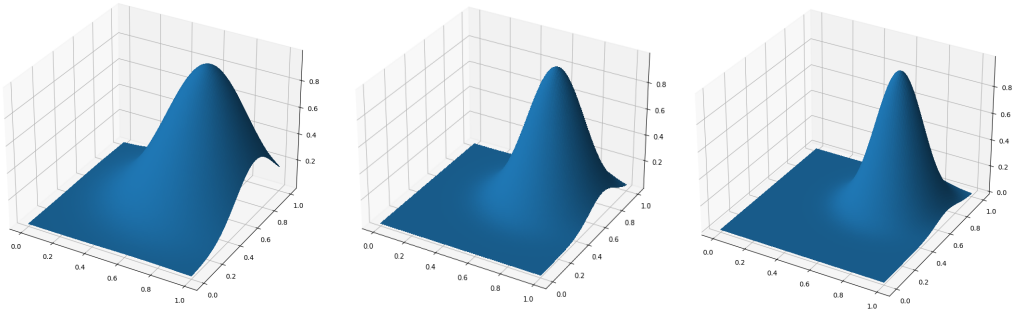


Figure 4.14: The three gaussian function with parameters 10, 20 and 30 plotted.

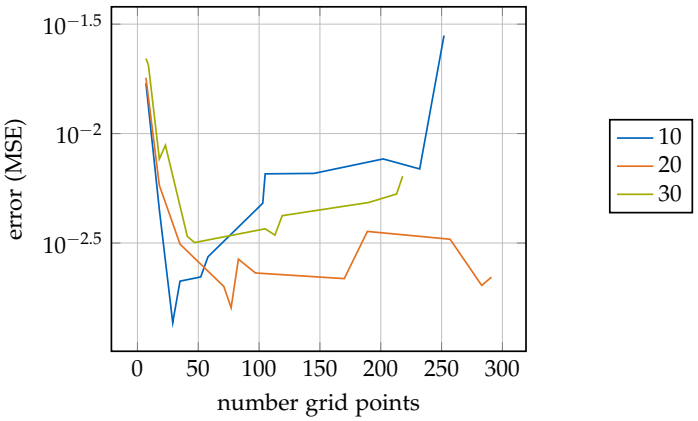


Figure 4.15: Testing error depending on the number of grid points (evaluations) plotted for the three gaussian functions.

We can observe in this plot that up to a distinct number of gird points, the error of the regression decreases. Until this point, the function can be reconstructed well. A higher number of evaluations then leads to an overfitted grid that can only predict the training data well. The error of the test set increases again like in figure 4.15.

An example for different refinement steps of the case with the function with parameter 20 in each dimension can be seen in figure 4.16.

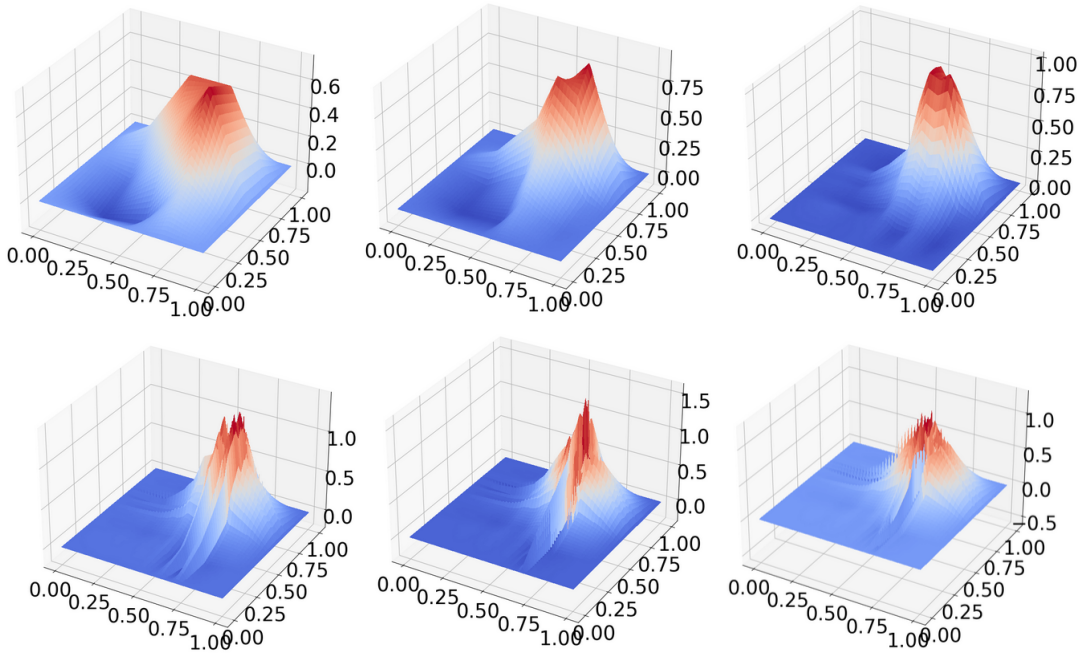


Figure 4.16: Three different refinement steps for the gaussian function with standard deviation 20. Number of grid points are 7 (left), 18 (center) and 52 (right) in the top row. In the bottom row the number of evaluations are 149, 255 and 297 (left, center and right respectively) where the effect of overfitting is getting more significant.

The number of evaluations are 7, 18, 52 in the top row (left, center, and right plot). In the bottom row, the number of evaluations are 149, 255, and 297 (left, center and right respectively). It can be observed that the function gets reconstructed better with increasing number of evaluations until a certain border point (52). If the number gets bigger than that, the result gets worse and the error increases. The bottom row of figure 4.16 shows this behavior.

4.2.2 Regularization parameter and term

The next parameter that we want to evaluate is the regularization parameter λ. In the following experiments, the impact of the parameter on the accuracy of the regression with the spatially adaptive Combination Technique is of interest. In chapter 2, it was

already mentioned that it is responsible for the trade off between smoothness of the function and exactness. To observe its impact, four different data sets are used. The first two being the boston housing and diabetes sets from the `sklearn` library. They were already used in previous experiments. The second two are samples of the discontinuous function and the gaussian function. For the latter ones, the dimension is set to 6, and 1000 random data points are taken. Additional white noise is added to the target values of the last two data sets. The discontinuous function has parameters $(2, 2, 2, 2, 2, 2)$ and $(0.7, 0.7, 0.7, 0.7, 0.7, 0.7)$. The maximum of the gaussian function is centered at $(0.5, 0.5, 0.5, 0.5, 0.5)$ and it has parameters 3 in each dimension.

For each test, 20% of the data set is taken as test data, and margin is set to 0.7. The refinement stops when the error is smaller or equal to the tolerance of 10^{-3} or the number of evaluations is higher than 200. As a regularization matrix, I is chosen in each case. The results can be seen in figure 4.17.

All four data sets have one thing in common. First, the error decreases with λ getting higher. And after reaching its own minimum, the error increases again without getting as small as the minimum again. In general, a regularization that is too small leads to an overfitted solution where unknown data points from the test set can not be predicted very well. In this case, the solution is adapted too much to the training set.

For both data sets in the top row (boston and diabetes from `sklearn`), the smallest error is reached with $\lambda = 10^{-8}$. For the other two cases in the bottom row, the optimal value is slightly higher, i.e. at 10^{-4} and 10^{-5} respectively. This is again dependent on the concrete data set used and the reason why this parameter has to be optimized for each regression task.

In general, we can observe that the optimal λ is smaller for higher dimensional data sets. The reason therefore is that the values of the system matrix B get smaller. A higher value for the regularization parameter would lead to a dominating matrix I or C and that would increase the overall error.

4.2.3 Different Opticom approaches

The use of the Combination Technique in the spatially adaptive case leads room to optimization of the coefficients. Just like in the previous case without refinement, the second step of the pipeline which is the Opticom leads to better coefficients of the component grids after the regression was already performed. These optimized hyperparameters then lead to a better overall solution and to a smaller error in the predictions of test data.

In the following, the three different Opticom approaches are compared. The focus lays on the accuracy and the time used to compute suitable values. As a data set, 700

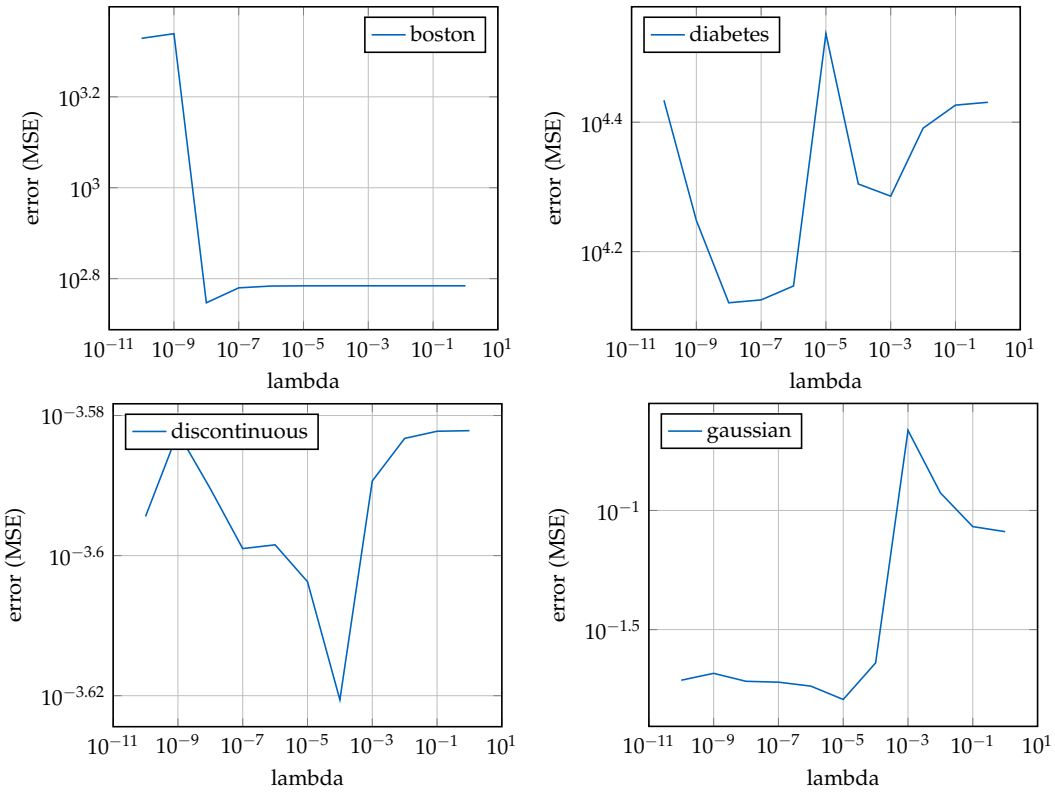


Figure 4.17: Error of the test set depending on the regularization parameter λ for data set boston housing and diabetes from `sklearn` library and samples for the two functions `discontinuous` and `gaussian`.

samples of the oszillatory function with additional white noise are used. The plot of the function and the samples can be seen in figure 4.18. The parameter `max_evaluations` is set to 100 and margin is chosen as 0.7. The error tolerance is 10^{-3}. The regularization consists of an optimized λ and the matrix I. The comparison of the three Opticom approaches can be seen in table 4.4. Two tests were made for the first ansatz, one with $\lambda_{Opticom} = \lambda$ and the other one with an optimized parameter for the Opticom.

The accuracy of the regression with the first and second (Garcke and least squares based without regularization) approach is better than without Opticom. For the first one, the tuning of the regularization parameter brings further improvements of predictions. For the last approach which is not based on the data but rather only on the solutions of the component grids, the error of the test set increases. This is not always the case but here, the errors of the partial solutions already cancel out quite well leading to worse predictions when changing the coefficients.

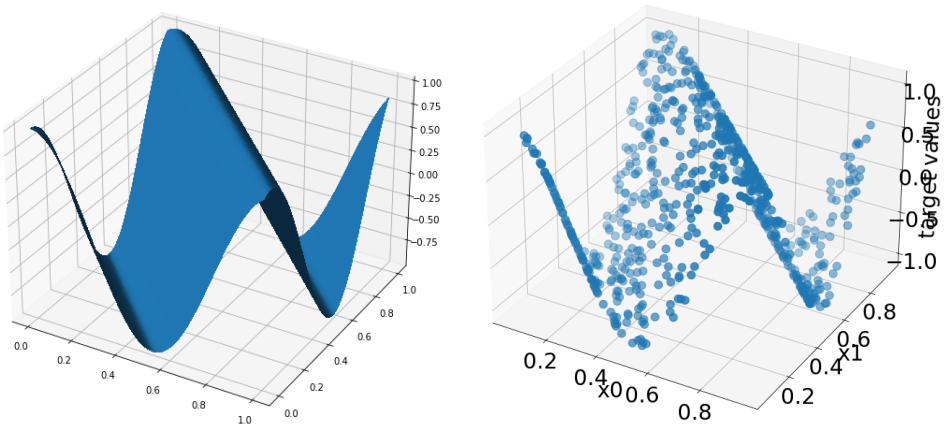


Figure 4.18: Oszillatory function (left) and samples (right) plotted.

opticom variant	error (MSE)	time of Opticom (s)
Without	0.03270	0.0
Garcke optimized	0.03196	2.40577
Garcke without optimization	0.03201	2.46801
Least squares based without regularization	0.02853	2.00643
Error based	0.03903	0.01922

Table 4.4: Comparison of the three Opticom variants regarding testing error and time used. For the first approach, the regularization parameter $\lambda_{Opticom}$ is optimized in one case.

The complexity of the first approach is the worst one. Especially in cases of a high number of evaluations, this leads to problems. This is especially the case for higher dimensional problems. The time needed for the second optimization mainly depends on the number of data points and is therefore not critical. The fastest approach is the third one, but because of the fact that it does not depend on the validation set makes this third approach the worst one in general. Because of the high complexity of the first approach, the next tests are made with the second one. In general, the first two approaches take much more time than the regression without Opticom needs. The computation of the weights of the basis functions on the component grids takes much less than one second (measured approximately 0.1s).

The next tests aim to compare the normal regression with the spatially adaptive Combination Technique with the one with optimized coefficients of the component grids. The results in table 4.6 show how the accuracy of predictions is improved. For all

four tests, the configurations can be seen in table 4.5. The column of Opticom indicates that for all four data sets, the second Opticom approach (least squares based without regularization) is used.

Data set	margin	max_evaluations	Opticom
Diabetes	0.6	400	2
Discontinuous	0.8	50	2
Gaussian	0.6	100	2
Oszillatory	0.5	50	2

Table 4.5: Configurations of the regression methods for the evaluation of Opticom in the spatially adaptive case.

data set	without Opticom	with Opticom	Reduction
Diabetes	13259.25	12915.17	3%
Discontinuous	0.00017504	0.00015756	10%
Gaussian	0.068371	0.047443	30%
Oszillatory	0.482316	0.476549	2%

Table 4.6: Comparison of the testing error with and without performing Opticom.

As the table shows, the errors decrease after the optimization. The percentage in the right column shows how big the impact of the Opticom is. It is calculated with $1 - \frac{\text{error with Opticom}}{\text{error without Opticom}}$. These values differ among the data sets. The reason for this is that with the Opticom, only the hyperparameters are tuned. If the solution is already very close to the actual function, then the improvement can not be that high. The high reduction of the error happens when the coefficients of the component grids in the normal Combination Technique are not yet suitable. This is the case where high values in the third column appear.

4.3 Comparison with common regression

The implementation of the regression with the (optimized) Combination Technique was described and evaluated so far with different parameter settings and data sets. In this chapter, the implementation is compared with other common implementations for regression. The already mentioned library `sklearn` also offers the possibility to perform regression with different implementations. These are the following:

- Linear Regression
- Polynomial Regression
- Neural Network

The first one is an ordinary linear regression technique. The polynomial regression is an extension of it with more parameters that are optimized. Nonlinear functions can be reconstructed better with it. The third one is a neural network and uses a multi-layer perceptron.

4.3.1 Normal Combination Technique

To compare the implementation using the Sparse Grid Combination Technique with the one from the library, two different functions were used. The first one is the polynomial function and the second one is the discontinuous function. Both can be seen in figure 4.19. For both functions, $l_{min} = 1$ and for the first one $l_{max} = 5$ and the latter one $l_{max} = 4$ for the Combination Technique. The degree of the polynomial regression method is set to the number of grid points used in the own implementation to set similar conditions in all methods. The neural network performs 500 iterations and contains one hidden layer. The results of the comparison can be seen in figure 4.20.

It can be observed that depending on the data set or the function used, different methods reconstruct this function best. In the left diagram, where samples of the polynomial function are used, the Sklearn Polynomial regression outperforms the other methods with fewer data points. The neural network then has a smaller error which is due to the increasing number of data points. The regression with the Sparse Grid Combination Technique and the optimized version perform worse than those two library implementations for this data set. The linear regression from Sklearn has in general the highest error. The reason for the high accuracy of the Polynomial regression is that the function used can be reconstructed very well because it is polynomial. For the neural network, it is important that it has many training points so that predictions

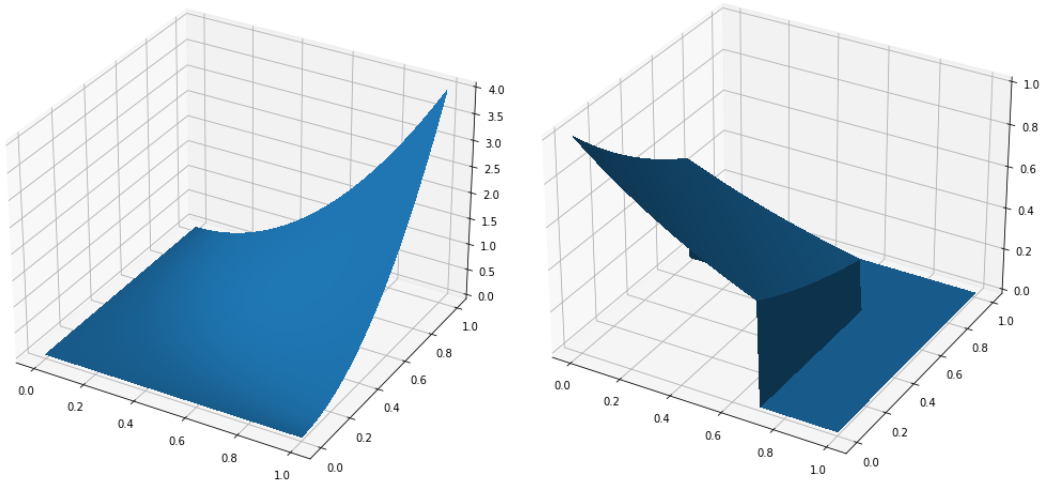


Figure 4.19: The polynomial function with coefficients $(2, 2)$ on the left and the discontinuous function on the right.

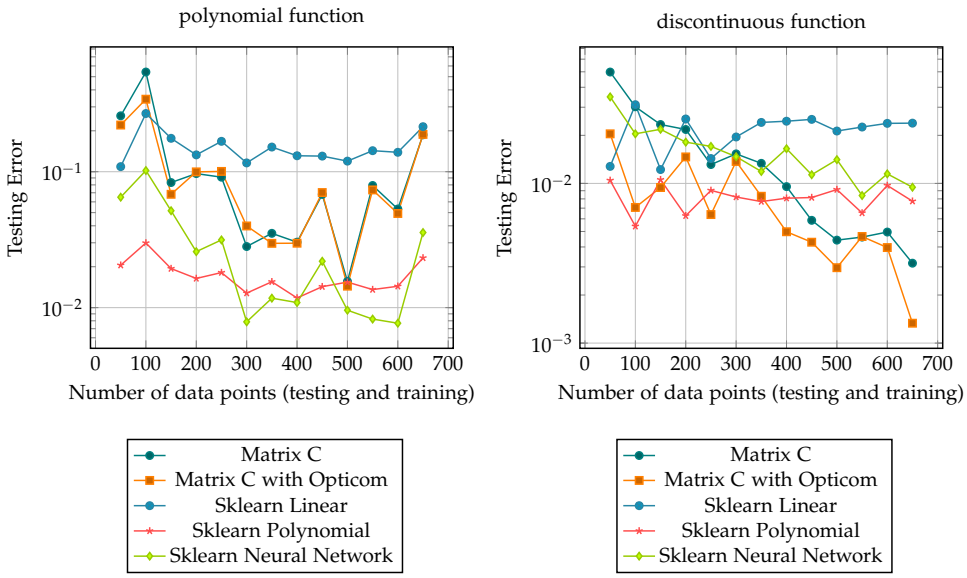


Figure 4.20: Testing Error (MSE) depending on the number of data points using the polynomial function (left) and the discontinuous function (right).

are more accurate. This is the reason why the error decreases very fast with growing number of data points.

For the second function, the best results are made with the regression with the

optimized Combination Technique followed by the one without Opticom. This time, the polynomial regression is always better than the neural network and the worst accuracy has the linear regression. The reason why the Sparse Grids implementation outperforms the other methods is that the function used to draw samples is discontinuous. This can not be reconstructed very well with polynomials. For the Sparse Grid implementation using weighted sums of hat functions, this is not a big problem and the testing errors are very low.

All in all, it really depends on the data set which implementation of regression has the lowest error.

4.3.2 Spatially adaptive Combination Technique

With the spatially adaptive Combination Technique, the regression improves compared to the normal one. In the following experiments, this version is included in the comparison. This time, the neural network and the polynomial regression from *sklearn* are used. Four different data sets are used. The gaussian function with maximum at 0.7 and parameters 20 in each dimension, the discontinuous function (discontinuous at $(0.8, 0.8, 0.8, 0.8)$ and parameters $(2, 2, 2, 2)$) and polynomial function with parameters 2 in each dimension and degree 2. All data samples of these data sets are four dimensional and no additional noise is added. We take 400 samples for each function. The fourth data set is the real-world data set California housing from the library *sklearn*. It has 20640 samples with dimension 8. For the own implementation, I is chosen as regularization matrix. In all cases, we optimize the regularization parameter λ to its optimum. The configurations can be seen in table 4.7. The number in the column of Opticom stands for the options described in chapter 3 (1: least squares based approach with regularization, 2: least squares based without regularization, 3: error based, 0: no Opticom).

	CT			adaptive CT		
Data set	l_{min}	l_{max}	Opticom	margin	max_evaluations	Opticom
Gaussian	1	5	2	0.7	300	2
Discontinuous	1	5	2	0.8	400	2
Polynomial	1	5	2	0.7	400	2
California housing	1	3	0	0.7	400	3

Table 4.7: Configurations of the regression methods for the comparison with the ones from sklearn.

The polynomial regression is always adapted to the grid, i.e. the degrees of freedom are always chosen to be similar to the number of grid points of our regression method.

The results of the comparisons can be seen in figure 4.21.

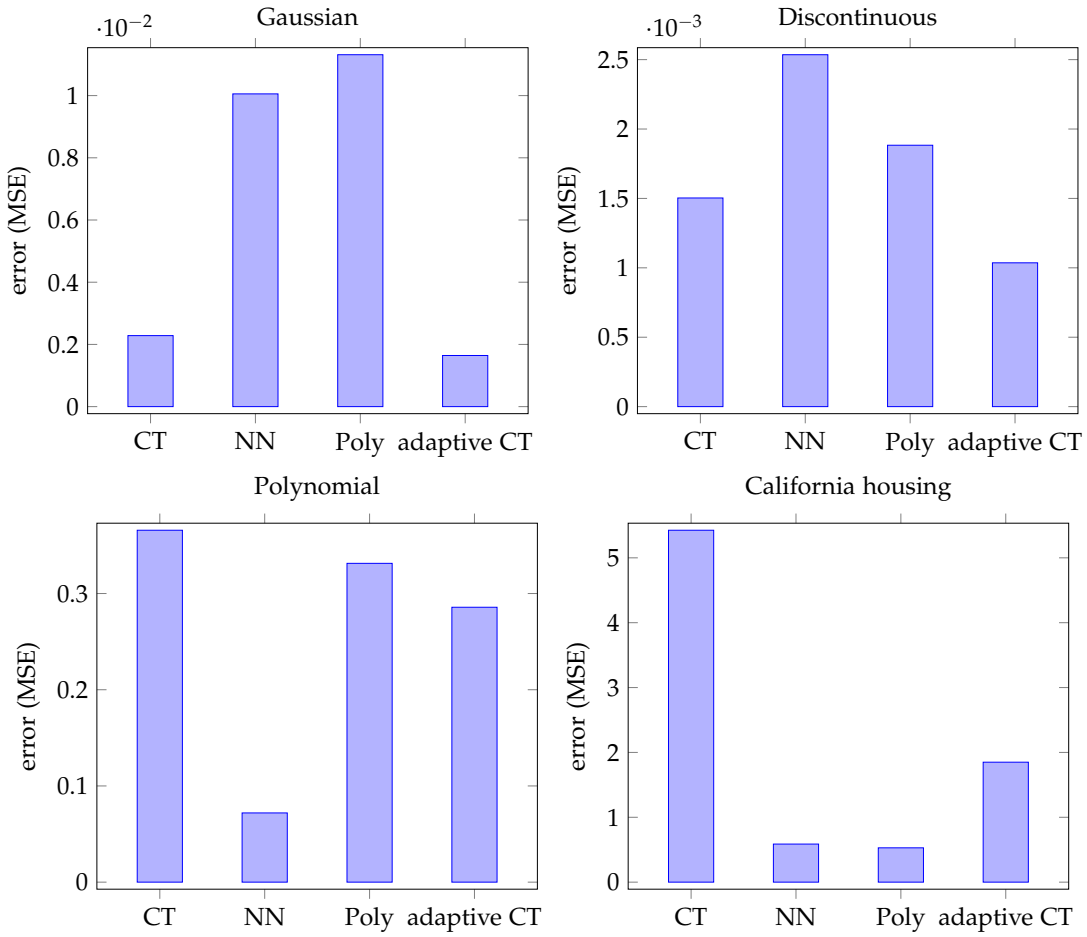


Figure 4.21: Comparison of the regression with the normal Combination Technique (CT), the neural network, the polynomial regression (Poly) and the adaptive regression with the spatially adaptive Combination Technique (adaptive CT). Data sets used are the gaussian, discontinuous and polynomial function and the real world data set California housing.

In all cases, the adaptive regression performs better than the one using the normal Combination Technique. For the gaussian and discontinuous functions, the adaptive regression with the spatially adaptive Combination Technique performs better than the two implementations from *sklearn*. For the polynomial function, the neural network achieves the smallest test error. In general, the regression with the Combination

Technique performs worse than the other implementations for the California housing data set. Notice that in all cases, the parameters for the polynomial regression had to be adapted and optimized to the data set. We do not have to choose a specific degree but only set the maximum number of points in the (adaptive) Combination Technique. This is one advantage of the regression with the spatially adaptive Combination Technique.

5 Conclusion and Outlook

In this work, many different possibilities for using the Combination Technique for regression were presented and evaluated. In the following, the results are summarized and an outlook on further improvements is given with some ideas for future work.

The simplest case of regression using the Combination Technique is based on the least squares problem. The first optimization is to add the regularization term with two different possibilities. We found out that C is slightly more accurate than matrix I, but with the second one being much less complex to compute, we came to the conclusion that I is the better choice. Additionally, the Combination Technique can be optimized with three different Opticom approaches. The two first ones are real optimizations using least squares and the third one updates it according to the error of the component grids. We observed that the approach introduced by Garcke ([8]) performs best with an optimized regularization parameter. Because of the complexity of this method, the second one based on the least squares problem optimizing the coefficients is used in most cases.

The next improvement is to use the spatially adaptive Combination Technique which refines depending on the data set. Again, a normal regression without regularization is introduced, followed by the two possibilities of I and C. Similar to the normal Combination Technique, the choice of using I is the best. Again, the three Opticom methods are presented, with the second option (based on least squares) being the best one for high dimensional data. The results of the experiments showed that in certain cases, the implementation can produce better results than common regression methods from the library *sklearn*.

Of course, there is still room for improvement. For example, the computation of the matrix C does not depend on the data set used but only on the grid. This brings the possibility of splitting the pipeline further into an offline and an online phase like it is done in [13]. In the first phase where no data is available, the matrix can be computed and stored. When the samples can be used in the online phase, the right matrix can then be fetched depending on the grid used. This tackles the problem of the high complexity of the computation of the matrix. For this purpose, it would be possible to implement a data base that stores different matrices for common scenarios.

Another optimization is to further improve the error based Opticom method. This is the fastest one only depending on the errors of the component grids. By comparing the error of the combined solution, a decision can be made telling whether the update of the coefficients should be performed or not. This can prevent a high increasing error. In general, this can not be applied to the other opticom methods because only the validation set is available in this phase and they use a real optimization which can not lead to worse errors regarding the validation set. In all cases, a reduced accuracy can be observed because of possible noise of the data and the fact that the overall error is measured with the test set which is not available to this time.

List of Figures

List of Tables

Bibliography

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

[2] D. Pflüger, "Spatially adaptive sparse grids for high dimensional problems," vol. 13, 2010.

[3] J. Garcke, "Sparse grids in a nutshell," in *Sparse Grids and Applications*, J. Garcke and M. Griebel, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 57–80, ISBN: 978-3-642-31703-3.

[4] J. Garcke, M. Griebel, and M. Thess, "Data mining with sparse grids," *Computing*, vol. 67, 2001. DOI: 10.1007/s006070170007.

[5] M. Griebel, M. Schneider, and C. Zenger, "A combination technique for the solution of sparse grid problems," 1990.

[6] M. Obersteiner and H.-J. Bungartz, "A generalized spatially adaptive sparse grid combination technique with dimension-wise refinement," *SIAM Journal on Scientific Computing*, vol. 43, no. 4, A2381–A2403, 2021. DOI: 10.1137/20M1325885. eprint: https://doi.org/10.1137/20M1325885.

[7] M. Hegland, J. Garcke, and V. Challis, "The combination technique and some generalisations," *Linear Algebra and its Applications*, vol. 420, no. 2, pp. 249–275, 2007, ISSN: 0024-3795. DOI: https://doi.org/10.1016/j.laa.2006.07.014.

[8] J. Garcke, "Regression with the optimised combination technique," in *Proceedings of the 23rd International Conference on Machine Learning*, ser. ICML '06, Pittsburgh, Pennsylvania, USA: Association for Computing Machinery, 2006, pp. 321–328, ISBN: 1595933832. DOI: 10.1145/1143844.1143885.

[9] M. Fabry, "Spatially adaptive density estimation with the sparse grid combination technique," Masterarbeit, Technical University of Munich, Sep. 2020.

[10] C. C. Moser, "Machine learning with the sparse grid density estimation using the combination technique," Bachelorarbeit, Technical University of Munich, Sep. 2020.

[11] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. DOI: 10.1038/s41586-020-2649-2.

[12] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, *et al.*, "Scikit-learn: Machine learning in python," *the Journal of machine Learning research*, vol. 12, pp. 2825–2830, 2011.

[13] 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. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9781611973440.51.