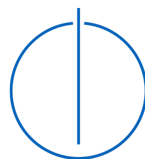
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Machine Learning with the Sparse Grid Density Estimation using the Combination Technique

Cora Charlotte Moser



DEPARTMENT OF INFORMATICS

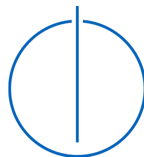
TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Machine Learning with the Sparse Grid Density Estimation using the Combination Technique

Maschinelles Lernen mit der Dünngitter Dichteschätzung unter Verwendung der Kombinationstechnik

Author:	Cora Charlotte Moser
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	M.Sc. Michael Obersteiner
Submission Date:	September 15, 2020



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

Munich, September 15, 2020 Cora Charlotte Moser

Acknowledgments

I want to acknowledge

- . . . my advisor, Michael Obersteiner, for his continuous support throughout the whole process of creating this thesis and the associated implementation,
- . . . my good friend, Hendrik Möller, for his help in frequently proofreading this thesis, as well as being available for all LaTeX-related questions at any given time,
- . . . all my other friends and family for their love and support.

Abstract

This thesis depicts, how density based supervised and unsupervised machine learning in the form of classification and clustering can be performed with sparse grids, which were built using the combination technique. Therefore the *sparseSpACE* framework, an environment for computing arbitrary operations with the combination technique, was expanded by a wrapper, which provides an interface for density based machine learning and integrates the already existing implementation of density estimation.
Classification wants to learn, which points of an input set are more likely to accept a certain class to then label unknown new data accordingly. Clustering on the other hand labels completely unknown input based on similar properties between points. To accomplish that, a "nearest-neighbor-graph" connecting most of the set is built and then reduced according to the estimated density, constructing new clusters. Such high-dimensional machine learning tasks are usually infeasible to compute with full grid density estimation, so the use of a sparse grid based approach is utilized. The implemented algorithms for both machine learning tasks are evaluated by testing and analyzing various sets of data.

Diese Arbeit beschreibt, wie überwachtes und unüberwachtes maschinelles Lernen im Sinne von Klassifikation und Clustering mithilfe der mit Kombinationstechnik erstellten Dünngittern ausgeführt werden kann. Dazu wurde das *sparseSpACE*-Framework, welches eine Umgebung für Berechnungen mit der Kombinationstechnik bietet, um einen Wrapper erweitert, der eine Schnittstelle für dichtebasiertes maschinelles Lernen bietet und die bereits vorhandene Dichteschätzungs-Funktionalität integriert.
Die Klassifizierung will anhand eines Eingabedatensatzes lernen, welche Punkte darin welche Klassen annehmen, um so unbekannte Daten zu klassifizieren. Clustering hingegen kategorisiert Datensätze komplett neu anhand von erkannten Ähnlichkeiten zwischen den Punkten. Zu diesem Zweck wird ein "Nearest-Neighbor-Graph" erstellt, der die meisten Punkte miteinander verbindet und der mithilfe der Dichtefunktion so reduziert wird, dass neue Kategorien entstehen. Solch mehrdimensionales maschinelles Lernen is normalerweise ungeeignet für Berechnungen mit Vollgittern, weshalb ein Dünngitteransatz benutzt wird. Die implementierten Algorithmen für beide Arten des maschinellen Lernens sind mit mehreren Datensätzen getestet und analysiert worden.

Contents

Acknowledgments iii

Abstract iv

List of Notations vii

1 Introduction 1

2 Theoretical Background 2
2.1 Full Grids 2
2.1.1 Nodal basis 3
2.1.2 Hierarchical basis 7
2.2 Sparse Grids 12
2.2.1 Combination Technique 13
2.2.2 Adaptive refinement 16
2.3 Machine Learning with Sparse Grids 17
2.3.1 Density Estimation 18
2.3.2 Classification 21
2.3.3 Clustering 23

3 Implementation 26
3.1 The sparseSpACE-framework 26
3.2 The DEMachineLearning wrapper 27
3.2.1 The Classification class 27
3.2.2 The Clustering class 30

4 Results 36
4.1 Classification 37
4.1.1 Full vs. Sparse grids 37
4.1.2 Minimal vs. Maximal Level 45
4.1.3 Static data sets 46
4.1.4 Under- and Overfitting 48
4.1.5 Learning to testing ratio 48

4.2 Clustering 50
4.2.1 Full vs. Sparse grids 50
4.2.2 Clustering in more dimensions 56
4.2.3 Varying number of nearest neighbors 57
4.2.4 Varying cutting threshold 58

5 Conclusion **59**
5.1 Summary 59
5.2 Outlook 59

List of Figures **61**

List of Tables **66**

Bibliography **67**

List of Notations

Notation	Description
$\mathbb{R}$	All real numbers including 0.
$\mathbb{N}$	All natural numbers excluding 0.
$\mathbb{N}_0$	All natural numbers including 0.
$\mathcal{O}$	Big-$\mathcal{O}$ notation.
$\vec{a}$	The vector $\vec{a}$.
$\boldsymbol{A}$	The matrix $\boldsymbol{A}$.
$\vec{K}$	Vector with all entries being the constant K.
$\circ$	Substitution for binary operators $=$, $\leq$, $\geq$, $+$, $-$.
$\vec{a} \circ \vec{b}$	The binary operator $\circ$ is applied component-wise to vectors $\vec{a}$ and $\vec{b}$.
$\Phi(x)$	The standard hat function.
$\bar{\Omega}$	Domain of all grid points in one dimension.
$\bar{\Omega}^d$	Domain of all grid points in d dimensions.
ℓ	Discretization level of a one-dimensional grid.
$\vec{\ell}$	Discretization level vector of a d-dimensional grid.
h_ℓ	Mesh size of a ℓ-level grid.
$\vec{h}_{\vec{\ell}}$	Mesh size vector of a $\vec{\ell}$-level grid.
$x_{\ell,i}$	Grid point at index i of a ℓ-level grid.
$\vec{x}_{\vec{\ell},\vec{i}}$	Grid point at index vector $\vec{i}$ of a $\vec{\ell}$-level grid.
$\Phi_{\ell,i}(x)$	Hat basis function over grid point $x_{\ell,i}$.
$\Phi_{\vec{\ell},\vec{i}}(\vec{x})$	Hat basis function over grid point $x_{\vec{\ell},\vec{i}}$.
V_ℓ	Function space of a ℓ-level full grid.
$V_{\vec{\ell}}$	Function space of a $\vec{\ell}$-level full grid.
W_k	Sub function spaces of V_ℓ with $1 \leq k \leq \ell$.
$W_{\vec{k}}$	Sub function spaces of $V_{\vec{\ell}}$ with $\vec{1} \leq \vec{k} \leq \vec{\ell}$.
$U \bigoplus V$	The direct sum of sets U and V.
$u_\ell(x)$	Interpolation function based on a ℓ-level full grid.
$u_{\vec{\ell}}(\vec{x})$	Interpolation function based on a $\vec{\ell}$-level full grid.

Notation	Description
$\lvert\vec{\ell}\rvert_1$	Norm, which describes the sum of a level vector $\vec{\ell}$.
$\lvert\vec{\ell}\rvert_\infty$	Norm, which describes the maximal level ℓ_i of the level vector. $\vec{\ell}$
$V_n^{(s)}$	Function space of a hierarchical sparse grid with level n in all dimensions.
ℓ_{min}	The minimal level of a combination-technique sparse grid.
ℓ_{max}	The maximal level of a combination-technique sparse grid.
$\Omega^{(c)}_{\ell_{\text{min}},\ell_{\text{max}}}$	Combination-technique sparse grid.
$V^{(c)}_{\ell_{\text{min}},\ell_{\text{max}}}$	Function space of a combination-technique sparse grid.
$\Omega_{\vec{\ell}}$	Component grid of a combination-technique sparse grid.
$V^{(c)}_{\vec{\ell}}$	Function space of a component grid.
$u_n^{(s)}(\vec{x})$	Interpolation function based on a hierarchical sparse grid.
$u^{(c)}_{\ell_{\text{min}},\ell_{\text{max}}}(\vec{x})$	Interpolation function based on a combi.-technique sparse grid.
$\lvert V_n\rvert$	Number of grid points of a grid with function space V_n.
$\lVert u-\tilde{u}\rVert_p$	L_p-error of the approximated function $\tilde{u}$.
$\hat{f}_{\text{kernel}}(x)$	Kernel-approximated density function.
$\hat{f}_{\text{grid}}(\vec{x})$	Grid-approximated density function.
$\hat{f}_{\text{sgrid},N}(\vec{x})$	Sparse-grid-approximated density function with N total grid points.
$(\cdot\,,\,\cdot)_{L^2}$	L^2 inner product of a pair of basis functions.
D_{train}	Training set of a classification problem.
C	Number of classes for a classification problem.
$g(\vec{x})$	Mapping function of a classification problem.
$\hat{g}(\vec{x})$	With density estimation approximated mapping function.
$D_{\text{train},j}$	Subset of D_{train} with every class being j.
$\hat{f}_j(\vec{x})$	With set $D_{\text{train},j}$ approximated density function.
D	Input set for a clustering problem.
t_d	Density threshold of a clustering problem.
$R(t_d)$	All input points in D with a density above t_d.
t_c	Edge cutting threshold of a clustering problem.
$e_{i,j}$	Edge between two points $\vec{x}_i$ and $\vec{x}_i$.
$\vec{x}_{\text{mid}(i,j)}$	Exact midpoint of two points $\vec{x}_i$ and $\vec{x}_i$ on $e_{i,j}$.
$\hat{f}_{\text{mid}(i,j)}(\vec{x}_i,\ \vec{x}_j)$	Density estimation function evaluating the density of $\vec{x}_{\text{mid}(i,j)}$.

1 Introduction

Machine Learning today is a multifarious field in computer science. With the era of *big data*, which describes huge amounts of data not suited for manual processing, the need for automated methods of data analysis arises [1]. Those methods care to not strictly memorize patterns in data, but to learn them and understand their general rules. Machine Learning is therefore usually divided into the two main categories *supervised* and *unsupervised learning*, and sometimes also into the third minor category *reinforcement learning*. The former two are often described in the context of *classification* and *clustering*.
There are various ways to perform machine learning tasks on some sets of data, one possibility is with the use of *estimated density functions*. Those can be constructed with the use of *grids*. A *regular full grid* with d dimensions and n equidistant grid points in every dimension, i.e. points with the same spacing, contains a total of n^d grid points, whereas the grid-dimension is dependent on the data-set-dimension. This is the cause of a significant problem: The number of grid points on a full grid and therefore the complexity of the machine learning task increases exponentially with the dimension d. This problem is often described as the *curse of dimensionality*.
To tackle this issue Zenger presented in 1991 the approach of *sparse grids* [2], which efficiently reduces full grids to keep as much information as possible while also forcing a low complexity and error.
Zenger, Griebel and Schneider expanded on this in 1992 [3] and presented the *combination technique*, which is an alternative method to construct sparse grids and more importantly easier to implement than any previous approaches.
Chapter 2 provides a basic understanding about full and sparse grids and presents approaches to perform classification and clustering with sparse-grid-based density estimation. Then in chapter 3 the algorithms and procedures for the implementation, which was created in the scope of this thesis, are explained. Results of this implementation are described in chapter 4. Lastly chapter 5 summarizes the implementation and all its achieved results and provides suggestions for further research.

2 Theoretical Background

This chapter provides step-by-step insights into the interpolation on *full grids* and *sparse grids*, *density estimation*, *classification* and *clustering*. As part of sparse grids especially the *combination technique* is explained, as well as a brief introduction to *adaptive refinement* is given.

2.1 Full Grids

A one-dimensional full grid possesses n points x_i with $i \in [0,\ n-1]$, which each is used as the input of one *basis function* $b_i(x)$ of a *family of basis functions*. Each output is then factored by a *weight* w_i and summed up to the interpolation function $u(x)$:

$$u(x) := \sum_{i=0}^{n-1} w_i \cdot b_i(x) \tag{2.1}$$

There are many families of basis functions to choose from, each having certain properties suitable for different applications. In section 2.1.1 and section 2.1.2 two approaches are discussed, the *nodal basis* and the *hierarchical basis*. In the context of this thesis both are based on the well known linear *hat function* $\Phi(x)$ (see Figure 2.1):

$$\Phi(x) := \max\left(1 - |x|\,,\ 0\right) \tag{2.2}$$

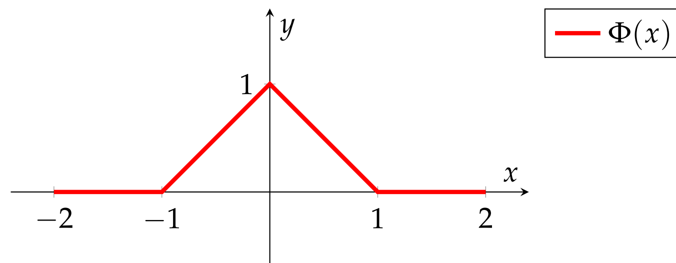


Figure 2.1: The standard hat function $\Phi(x)$ plotted in the domain $[-2,\ 2]$.

2.1.1 Nodal basis

For simplification purposes the problem of constructing the *nodal basis* is first abstracted to one dimension then expanded to d dimensions in this section.

One-dimensional nodal basis

The one dimensional unit interval $\bar{\Omega} := [0,\ 1]$ is considered to be the domain of all grid points. The *discretization level* ℓ of a grid divides it into a mesh with ℓ equidistant parts, which each have a mesh size of $h_\ell := 2^{-\ell}$ [4]. There are two different ways to sample the grid points, with or without boundaries. For the sake of simplicity only the case without boundary points is discussed. If one were to consider the boundary points, adjustments to the boundary basis functions would have to be made later. With this the grid points are indexed as follows:

$$x_{\ell,i} := i \cdot h_\ell, \quad 0 \le i \le 2^\ell \qquad \text{with boundary points} \tag{2.3}$$

$$x_{\ell,i} := i \cdot h_\ell, \quad 1 \le i \le 2^\ell - 1 \qquad \text{without boundary points} \tag{2.4}$$

Because the grid boundary points are omitted in all methods of this chapter, the number of sample points in a grid with level ℓ is always exactly $2^\ell - 1$.
To get a family of basis functions with the set of indices $I_\ell := \left[1,\ 2^\ell - 1\right]$ to interpolate on a grid with level ℓ, the standard hat function is applied to every sample point so that its maximum is right above the center of a mesh piece. Therefore the basis function $\Phi_{\ell,i}(x)$ for grid point x_i has the support $[x_{\ell,i} - h_\ell,\ x_{\ell,i} + h_\ell]$ and one basis function with index i can be rewritten as:

$$\Phi_{\ell,i}(x) := \Phi\left(\frac{x - x_{\ell,i}}{h_\ell}\right) = \max\left(1 - \left|\frac{x - x_{\ell,i}}{h_\ell}\right|,\ 0\right) \tag{2.5}$$

This family of basis functions is usually called *nodal* or *Lagrange basis* [4]. Using these basis functions the function space V_ℓ (see Figure 2.2) is defined as:

$$V_\ell := \operatorname{span}\left\{\Phi_{\ell,i} \mid i \in I_\ell\right\} \tag{2.6}$$

V_ℓ can then be used to interpolate the function $u_\ell(x)$ on a ℓ-level 1D grid by weighting every basis hat function $\Phi_{\ell,i}(x)$ by a weight factor $w_{\ell,i}$ and summing them up afterwards (see Figure 2.3):

$$u_\ell(x) := \sum_{i \in I_\ell} w_{\ell,i} \cdot \Phi_{\ell,i}(x) \tag{2.7}$$

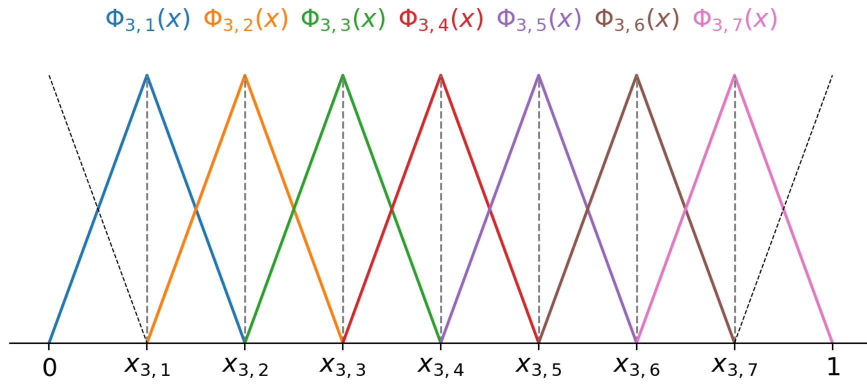


Figure 2.2: Example of nodal basis function space V_3 with level 3 and hat functions $\Phi_{3,i}(x)$ on grid points $x_{3,i}$ with $i \in I_3 := [1,\ 7]$. The boundary points are indicated with thin dotted black lines on the left and right borders.

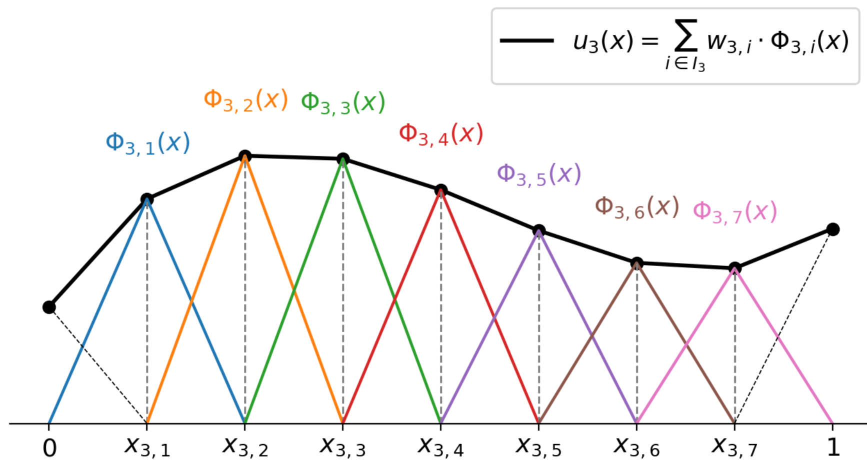


Figure 2.3: Example of a level 3 grid interpolated function $u_3(x)$ with weights $w_{3,i}$ (indicated by vertical gray dotted lines) and the nodal basis function space V_3. The boundary points are indicated with thin dotted black lines on the left and right borders.

Multi-dimensional nodal basis

When adapting the one dimensional nodal basis approach to d dimensions, the domain of all grid points turns into $\bar{\Omega}^d := [0,\ 1]^d$ and the level and indices adapt into vectors $\vec{\ell}$ and $\vec{i}$ with d entries each:

$$\vec{\ell} := (\ell_1,\ \ldots,\ \ell_d) \qquad \text{The level of the grid in every dimension} \tag{2.8}$$

$$\vec{i} := (i_1,\ \ldots,\ i_d) \qquad \text{The index of a grid point in every dimension} \tag{2.9}$$

A grid is separated into a mesh with equidistant hyper rectangles with varying mesh size in each dimension. So the mesh size also needs to be defined as a vector:

$$\vec{h}_{\vec{\ell}} := \left(h_{\ell_1},\ \ldots,\ h_{\ell_d}\right) = 2^{-\vec{\ell}} \tag{2.10}$$

As the boundary points are not considered, the set of index vectors is expanded to

$$I_{\vec{\ell}} := \left\{\vec{i} \in \mathbb{N}^d \mid \vec{1} \leq \vec{i} \leq 2^{\vec{\ell}} - \vec{1}\right\} \tag{2.11}$$

with $\vec{1} := (1,\ \ldots,\ 1)$ and the vector of all grid points is defined as (see Figure 2.4):

$$\vec{x}_{\vec{\ell},\vec{i}} := \left(x_{\ell_1,i_1},\ \ldots,\ x_{\ell_d,i_d}\right), \quad \vec{i} \in I_{\vec{\ell}} \tag{2.12}$$

For each of those grid points the basis function $\Phi_{\vec{\ell},\vec{i}}(x)$ is constructed as the tensor product of the linear one-dimensional basis function $\Phi_{\ell,i}(x)$ in each dimension with the support $\prod_{j=1}^{d}\left[x_{\ell_j,i_j} - h_{\ell_j},\ x_{\ell_j,i_j} + h_{\ell_j}\right]$ (see Figure 2.5):

$$\Phi_{\vec{\ell},\vec{i}}(\vec{x}) := \prod_{j=1}^{d} \Phi_{\ell_j,i_j}(x_j) \tag{2.13}$$

Analogous to the one-dimensional approach (see equation 2.6) the function space $V_{\vec{\ell}}$ is constructed as follows:

$$V_{\vec{\ell}} := \text{span}\left\{\Phi_{\vec{\ell},\vec{i}} \mid \vec{i} \in I_{\vec{\ell}}\right\} \tag{2.14}$$

And therefore the multi-dimensional interpolation function $u_{\vec{\ell}}(\vec{x})$ is defined as:

$$u_{\vec{\ell}}(\vec{x}) := \sum_{\vec{i} \in I_{\vec{\ell}}} w_{\vec{\ell},\vec{i}} \cdot \Phi_{\vec{\ell},\vec{i}}(\vec{x}) \tag{2.15}$$

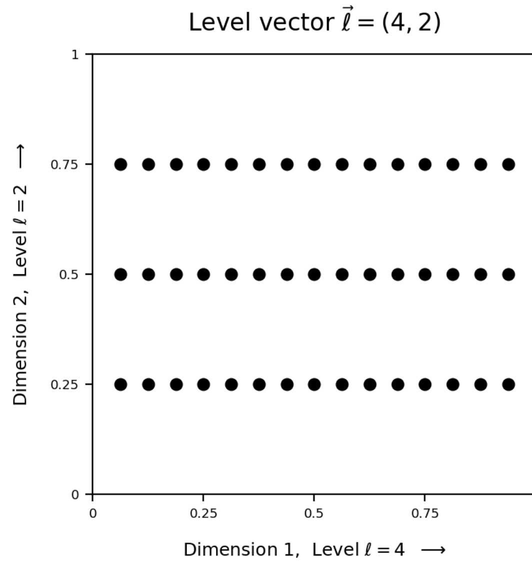


Figure 2.4: Example of a 2-dimensional grid with level 4 in the first and level 2 in the second dimension. The grid points $x_{(4,2),\vec{i}}$ have the index set $I_{(4,2)} := ((1,\ 1),\ \ldots,\ (15,3))$.

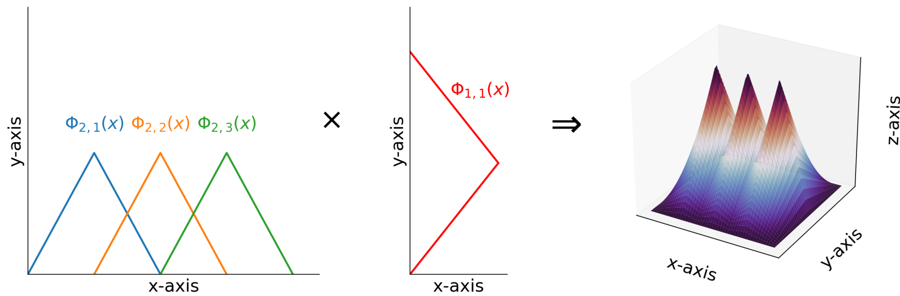


Figure 2.5: Example of the tensor product approach to generate the 3D base functions $\Phi_{(2,1),(1,1)}(x)$, $\Phi_{(2,1),(2,1)}(x)$ and $\Phi_{(2,1),(3,1)}(x)$ (right) with the 2D base functions $\Phi_{2,1}(x)$, $\Phi_{2,2}(x)$ and $\Phi_{2,3}(x)$ (left) of level 2 and the 2D base function $\Phi_{1,1}(x)$ (middle) of level 1.

2.1.2 Hierarchical basis

The hierarchical approach is quite similar to the nodal approach. But instead of directly building the function space V_ℓ out of all basis functions with level ℓ in the current dimension, all previous function spaces W_k with $1 \leq k \leq \ell$ are build first with the key difference that the points from previous levels are omitted. Those W_k are then summed up to reconstruct the nodal space V_ℓ [4].
As with the nodal approach, the one-dimensional case is discussed first and then expanded to d dimensions.

One-dimensional hierarchical basis

The domain of all grid points in one dimension is again $\bar{\Omega} := [0,\ 1]$ and the discretisation level ℓ again defines the mesh size $h_\ell := 2^{-\ell}$ of a level ℓ grid. But in contrary to the nodal approach, only every sample point with an odd index is considered. The boundary sample points are dismissed and the grid points are indexed as follows:

$$x_{\ell,i} := i \cdot h_\ell, \quad 1 \leq i \leq 2^\ell - 1 \quad i \text{ odd} \tag{2.16}$$

So the number of sampled points on a grid of level ℓ will always be $2^{\ell-1}$. With this the new index set is

$$I_{\text{odd},\ell} := \left\{ i \in \mathbb{N} \mid 1 \leq i \leq 2^\ell - 1,\ i \text{ odd} \right\} \quad , \tag{2.17}$$

but the support of one basis function $\Phi_{\ell,i}(x)$ at grid point $x_{\ell,i}$ is still $[x_{\ell,i} - h_\ell,\ x_{\ell,i} + h_\ell]$ and the definition of the basis function itself also doesn't change (see equation 2.5). This family of basis functions with the new set of indices is called *hierarchical basis* [4]. Note that in this basis the individual basis functions are pairwise disjoint (see Figure 2.6).
To interpolate functions with this basis, the function space V_ℓ has to be created. Therefore every sub-function space W_k with $1 \leq k \leq \ell$

$$W_k := \text{span} \left\{ \Phi_{\ell,i} \mid i \in I_{\text{odd},\ell} \right\} \tag{2.18}$$

has to be created first. After that they can be all summed up to V_ℓ (see Figure 2.7):

$$V_\ell := \bigoplus_{1 \leq k \leq \ell} W_k \tag{2.19}$$

The corresponding interpolation function is constructed as follows (see Figure 2.8):

$$u_\ell(x) := \sum_{1 \leq k \leq \ell} \sum_{i \in I_{\text{odd},k}} w_{k,i} \cdot \Phi_{k,i}(x) \tag{2.20}$$

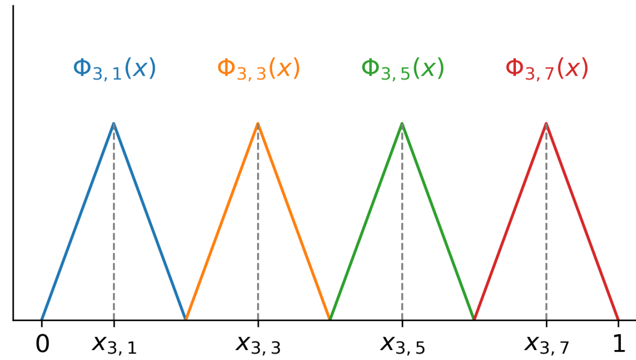


Figure 2.6: Example of hierarchical sub function space W_3 with level 3 and hat functions $\Phi_{3,i}(x)$ on grid points $x_{3,i}$ with $i \in I_{\text{odd},3} := [1,3,5,7]$.

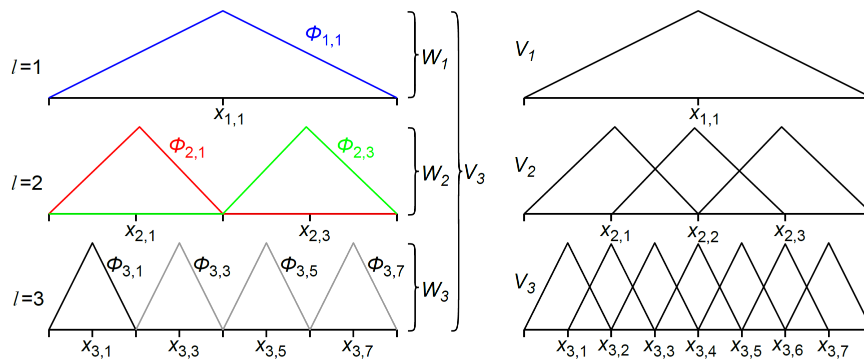


Figure 2.7: Side by side comparison of the construction of the function space V_ℓ with the nodal approach (right) and the hierarchical approach (left) [taken from 5].

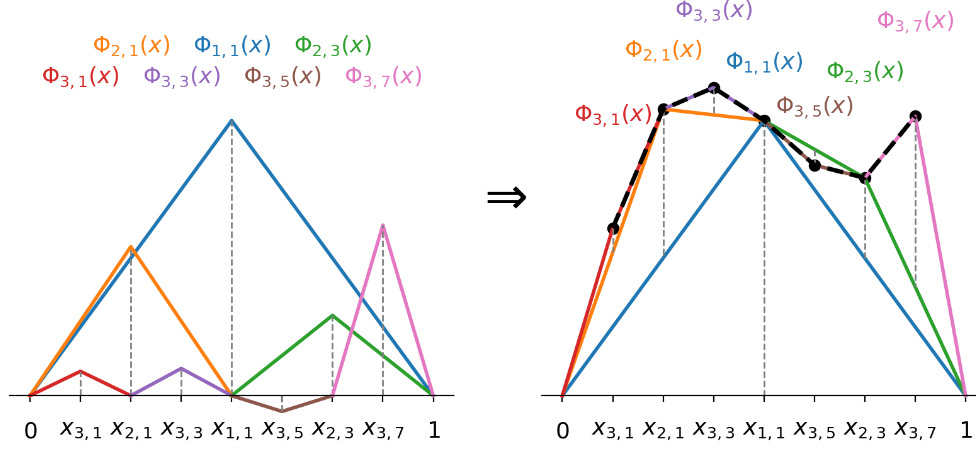


Figure 2.8: Example of a level 3 grid interpolated function $u_3(x)$ (indicated by the bold dotted black line) with weights $w_{3,i}$ (indicated by vertical gray dotted lines) and the hierarchical basis function space V_3 (right). The sub function spaces W_k with $k \in [1,\ 3]$ summed up together build the level 3 function space V_3 (left).

Multi-dimensional hierarchical basis

As with the adaptation of the nodal basis to d dimensions, the domain of all grid points becomes $\bar{\Omega}^d := [0,\ 1]^d$. Also level, indices and mesh size become vectors $\vec{\ell}$, $\vec{i}$ and $\vec{h}_{\vec{\ell}}$ accordingly with d entries each (see equation 2.8, equation 2.9 and equation 2.10). The index set however differs from the multi-dimensional nodal approach, because only every grid point with an odd index number is considered [4]. So the index set is defined as:

$$I_{\text{odd},\vec{\ell}} := \left\{ \vec{i} \in \mathbb{N}^d \mid \vec{1} \leq \vec{i} \leq 2^{\vec{\ell}} - \vec{1},\ \forall j \in \vec{i} : j \text{ odd} \right\} \tag{2.21}$$

With this the vector of all grid points is defined as:

$$\vec{x}_{\vec{\ell},\vec{i}} := \left(x_{\ell_1,i_1},\ \ldots,\ x_{\ell_d,i_d} \right), \quad \vec{i} \in I_{\text{odd},\vec{\ell}} \tag{2.22}$$

The basis function $\Phi_{\vec{\ell},\vec{i}}(\vec{x})$ for the multi-dimensional hierarchical basis again is constructed with the tensor product approach (see Figure 2.5) and has once again the support $\prod_{j=1}^{d} \left[x_{\ell_j,i_j} - h_{\ell_j},\ x_{\ell_j,i_j} + h_{\ell_j} \right]$ for each grid point. Therefore the definition of the multi-dimensional hierarchical basis function is the same as for the multi-dimensional nodal approach (see equation 2.13).

To construct the d dimensional function space $V_{\vec{\ell}}$, again every sub function space $W_{\vec{k}}$ with $\vec{1} \leq \vec{k} \leq \vec{\ell}$ has to be created first (see Figure 2.9):

$$W_{\vec{k}} := \operatorname{span}\left\{\Phi_{\vec{\ell},\vec{i}}(\vec{x}) \mid \vec{i} \in I_{\mathrm{odd},\vec{\ell}}\right\} \tag{2.23}$$

Summing all sub function spaces $W_{\vec{k}}$ up, the full function space $V_{\vec{\ell}}$ is obtained:

$$V_{\vec{\ell}} := \bigoplus_{\vec{1} \leq \vec{k} \leq \vec{\ell}} W_{\vec{k}} \tag{2.24}$$

The corresponding interpolation function is then defined as:

$$u_{\vec{\ell}}(\vec{x}) := \sum_{\vec{1} \leq \vec{k} \leq \vec{\ell}} \sum_{\vec{i} \in I_{\mathrm{odd},\vec{\ell}}} w_{\vec{k},\vec{i}} \cdot \Phi_{\vec{\ell},\vec{i}}(\vec{x}) \tag{2.25}$$

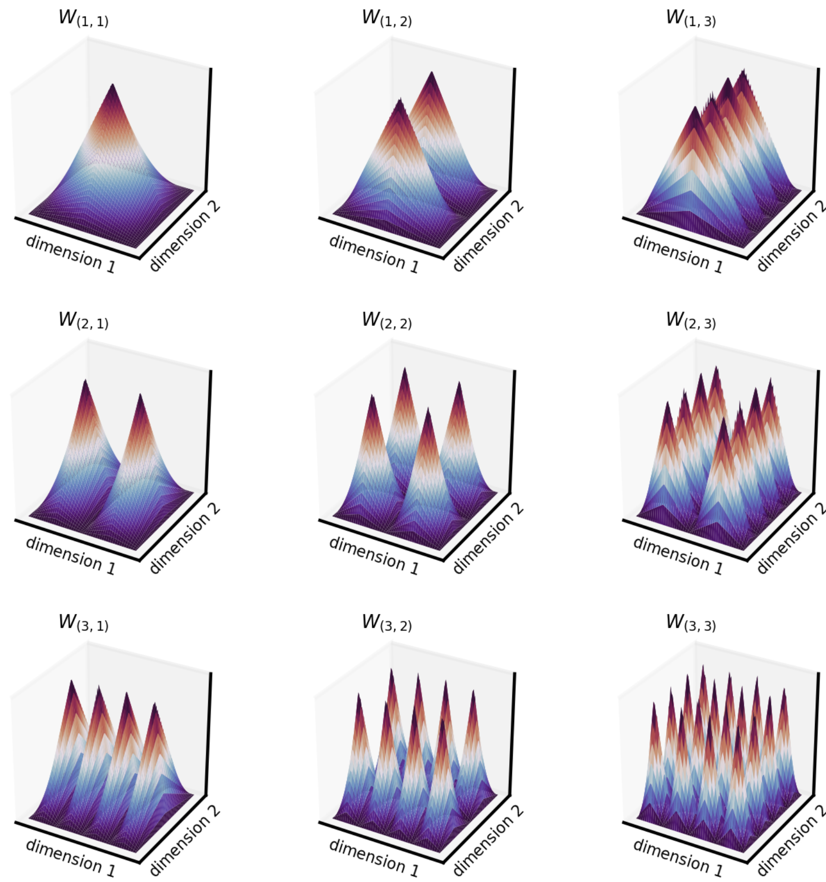


Figure 2.9: Example of the construction of all 2D sub function spaces $W_{\vec{k}}$ with $\vec{k} \in [(1,\ 1),\ (3,\ 3)]$ for the 2D function space $V_{(3,3)}$ with level 3 using the tensor product approach.

2.2 Sparse Grids

To convert *full grids* into *sparse grids* two definitions about the multi-dimensional level of a grid must be considered, the $|\vec{\ell}|_1$-norm and $|\vec{\ell}|_\infty$-norm [4]:

$$|\vec{\ell}|_1 := \sum_{j=1}^{d} |\ell_j| \qquad \text{sum of levels in all } d \text{ dimensions} \tag{2.26}$$

$$|\vec{\ell}|_\infty := \max_{1\leq j\leq d} |\ell_j| \qquad \text{the maximal level of all } d \text{ dimensions} \tag{2.27}$$

The hierarchical function space $V_{\vec{\ell}}$ can then be redefined as

$$V_n := V_{\vec{n}} = \bigoplus_{|\vec{\ell}|_\infty \leq n} W_{\vec{\ell}} \quad , \tag{2.28}$$

whereas $\vec{n} := (n, \ \ldots, \ n)$ describes a vector of length d with the entry n each. This means that every with this definition created hierarchical based function space has the same discretization level in every dimension.

This *full grid space* V_n can then be abstracted to the *sparse grid space* $V_n^{(s)}$ by only summing those $W_{\vec{k}}$ of V_n up, whose combined levels or rather the $|\vec{\ell}|_1$-norm are below the threshold $n + d - 1$ (see Figure 2.10):

$$V_n^{(s)} := \bigoplus_{|\vec{\ell}|_1 \leq n+d-1} W_{\vec{\ell}} \tag{2.29}$$

This leads to the interpolation functions u_n for the full and $u_n^{(s)}$ for the sparse grid space:

$$u_n(\vec{x}) := \sum_{|\vec{\ell}|_\infty \leq n, \, \vec{i}\in I_{\vec{\ell}}} w_{\vec{\ell},\vec{i}} \cdot \Phi_{\vec{\ell},\vec{i}}(\vec{x}) \tag{2.30}$$

$$u_n^{(s)}(\vec{x}) := \sum_{|\vec{\ell}|_1 \leq n+d-1, \, \vec{i}\in I_{\vec{\ell}}} w_{\vec{\ell},\vec{i}} \cdot \Phi_{\vec{\ell},\vec{i}}(\vec{x}) \tag{2.31}$$

To measure the efficiency of the sparse grid approach in comparison to the full grid approach, for both the amount of sample points is weighted up against the accuracy. The order of degrees of freedom or rather the number of grid points for the full and sparse grid spaces are then

$$|V_n| := \mathcal{O}\left(2^{nd}\right) \tag{2.32}$$

$$\left|V_n^{(s)}\right| := \mathcal{O}\left(2^n \cdot n^{d-1}\right) \quad , \tag{2.33}$$

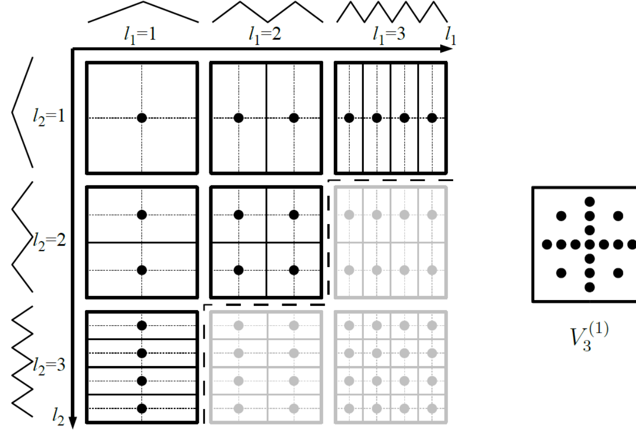


Figure 2.10: Example of the 9 sub spaces for levels $|\vec{\ell}|_\infty \leq 3$ (left, gray and black grids) which together form the full grid function space V_3 and the corresponding sparse grid space $V_3^{(s)}$ (right), which consists of all 5 sub spaces with levels $|\vec{\ell}|_1 \leq 4$ (left, black grids above the dashed line) [taken from 6].

which show a significant reduction in complexity for the sparse grid space in comparison with the full grid space. Meanwhile the estimated accuracy in the L_p-norm only slightly increases for the sparse grid space in comparison to the full grid space:

$$\|u - u_n\|_p := \mathcal{O}\left(4^{-n}\right) \tag{2.34}$$

$$\|u - u_n^{(s)}\|_p := \mathcal{O}\left(4^{-n} \cdot n^{d-1}\right) \tag{2.35}$$

So the sparse grid approach balances with increasing dimension d the rise in complexity and error by letting d influence both only in logarithmic terms. With this the *curse of dimensionality* can be overcome by some extent.

2.2.1 Combination Technique

The so called *combination technique* is another way to construct sparse grids by linearly combining certain independent anisotropic full grids between a *minimal level* $\ell_{\min}$ and a *maximal level* $\ell_{\max}$ [3]. That means to build a d-dimensional sparse grid $\Omega^{(c)}_{\ell_{\min},\ell_{\max}}$ in the function space $V^{(c)}_{\ell_{\min},\ell_{\max}}$ with the combination technique, all full grids $\Omega_{\vec{\ell}}$, also called *component grids*, with the function space $V^{(c)}_{\vec{\ell}}$ and the domain

$$L^{(c)} := \left\{\vec{\ell} \in \mathbb{N}^d \mid \forall \ell_j \in \vec{\ell} : \ell_{\min} \leq \ell_j \leq \ell_{\max},\ 1 \leq j \leq d\right\} \tag{2.36}$$

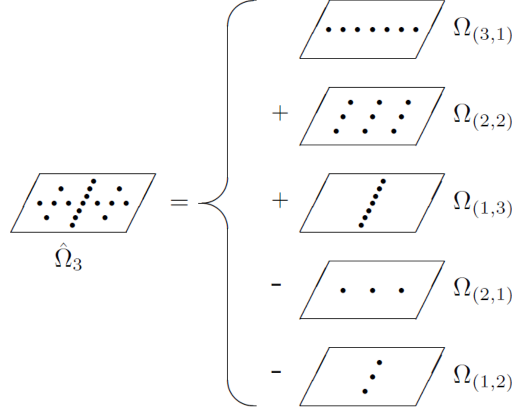


Figure 2.11: Example of a sparse grid $\Omega^{(c)}_{1,3}$ with $\ell_{\min} = 1$ and $\ell_{\max} = 3$ (left) constructed by the linear combination of five component grids. Those are added whenever $q = 0$ and therefore $\alpha_q = 4$ and subtracted whenever $q = 1$ and therefore $\alpha_q = 3$ (right) [taken from 4].

for their level vectors $\vec{\ell}$ are considered. For each component grid $\Omega_{\vec{\ell}}$ the corresponding interpolation function is $u_{\vec{\ell}}(\vec{x})$ (see equation 2.15 and equation 2.25).
To construct the sparse grid $\Omega^{(c)}_{\ell_{\min},\ell_{\max}}$, the sample points of every component grid $\Omega_{\vec{\ell}}$ are either added to or subtracted from the resulting sparse grid. The constraints for addition or subtraction are set by the iteration step or rather the hyperplane q with $q \in [0, d-1]$ and the $|\vec{\ell}|_1$-value of a component grid, which will furthermore represented with α_q:

$$|\vec{\ell}|_1 = \alpha_q \tag{2.37}$$
$$\alpha_q := \ell_{\max} + (\ell_{\min} \cdot (d-1)) - q \tag{2.38}$$

Every component grid $\Omega_{\vec{\ell}}$ with its $|\vec{\ell}|_1$-value α_q is added $\binom{d-1}{q}$ times for every even-valued q and analogously subtracted $\binom{d-1}{q}$ times for every odd-valued q (see Figure 2.11 and Figure 2.12). This is done in order to prevent sample points from overlapping with each other.
This now leads to the construction of the interpolation function [3]:

$$u^{(c)}_{\ell_{\min},\ell_{\max}}(\vec{x}) := \sum_{q=0}^{d-1} (-1)^q \cdot \binom{d-1}{q} \sum_{|\vec{\ell}|_1 = \ell_{\max} + (\ell_{\min} \cdot (d-1)) - q,\ \vec{\ell} \in L} u_{\vec{\ell}}(\vec{x}) \tag{2.39}$$

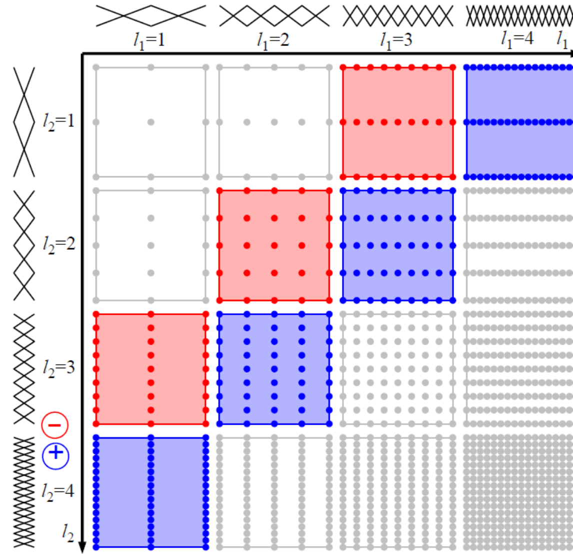


Figure 2.12: Example of all two-dimensional nodal based regular grids from level 1 to level 4 with boundaries. To create a sparse grid $\Omega^{(c)}_{1,4}$ with this function space, all blue component grids are added for the hyperplane 0 and all red component grids are subtracted for the hyperplane 1 [taken from 6].

Note that $\Omega^{(c)}_{\ell_{\min},\ell_{\max}}$ is just a regular full grid if $\ell_{\min} = \ell_{\max}$, because there is only one possible level vector in $L^{(c)}$ for that case.
When computing the order of degrees of freedom, for every component grid the number of partial solutions their size are combined:

$$\left|V^{(c)}_{\ell_{\min},\ell_{\max}}\right| := \mathcal{O}\left(d \cdot {\ell_{\max}}^{d-1}\right) \times \mathcal{O}\left(2^{\ell_{\max}}\right) \tag{2.40}$$

Because those partial solutions are computed on full grids, which there are significantly more applications for, the combination technique is usually simpler to implement than the hierarchical sparse grid approach and therefore widely used [6]. Also parallel implementation can be accomplished easily, because all component grids can be computed independently.
However, the L_p-error of the combination technique is roughly the same as the hierarchical sparse grid approach with:

$$\|u - u^{(c)}_{\ell_{\min},\ell_{\max}}\|_p := \mathcal{O}\left(4^{-\ell_{\max}} \cdot {\ell_{\max}}^{d-1}\right) \tag{2.41}$$

2.2.2 Adaptive refinement

A sparse grid can be *adaptively refined* in certain areas to shift computation accuracy to predominantly more or away from less important parts of the grid. This is useful for problems, which show significantly differing characteristics in two or more areas [6]. Also computation time can be reduced by mostly ignoring points, which contribute less to the overall solution, while simultaneously keeping the error as low as possible. In both so far discussed sparse grid approaches every area of a grid is refined evenly in one step of increasing the discretization level. With adaptive refinement only specific grid points can be selected for further refinement, e.g. based on the information of the local estimated error [6].
To achieve this, to all those selected grid points their respective children in the next hierarchical discretization level are added to the grid if they aren't already present. Because most algorithms on sparse grids traverse through the hierarchical structure, if points are added that can be reached though a different hierarchical route, those missing path points are added as well (see Figure 2.13).

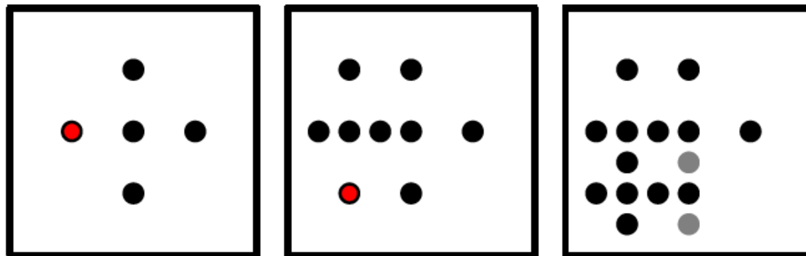


Figure 2.13: Example of three adaptive refinement steps of a sparse grid without boundaries. On a level 2 sparse grid, the most left point is selected for further refinement (left) and all its surrounding children added (middle). After yet another refinement step the missing path points (marked in gray) from previous discretization steps must also be added (right) [taken from 6].

2.3 Machine Learning with Sparse Grids

This section discusses how sparse grids (see section 2.2) can be used to approximate a density function on some data set and then with this perform supervised and unsupervised learning, here classification and clustering.
A density function lets the machine learning algorithm learn the structure of some input data by teaching it, where sample points are more or less likely to occur [7]. This in turn helps the algorithm derive how to operate on said data. The construction of this density function is affected by the curse of dimensionality for high-dimensional sets of data, hence sparse grids are a good approach to tackle this problem.
Two main problems arise with machine learning, *overfitting* and *underfitting* (see Figure 2.14). The first one describes the use of too many details being modeled and thus reducing the accuracy of the machine learning results, e.g. by valuing noise as much as relevant data. Underfitting is the exact opposite and occurs when relevant data is disregarded. In chapter 4, section 4.1.4 an explicit example will be given to provide a better understanding of those two terms.

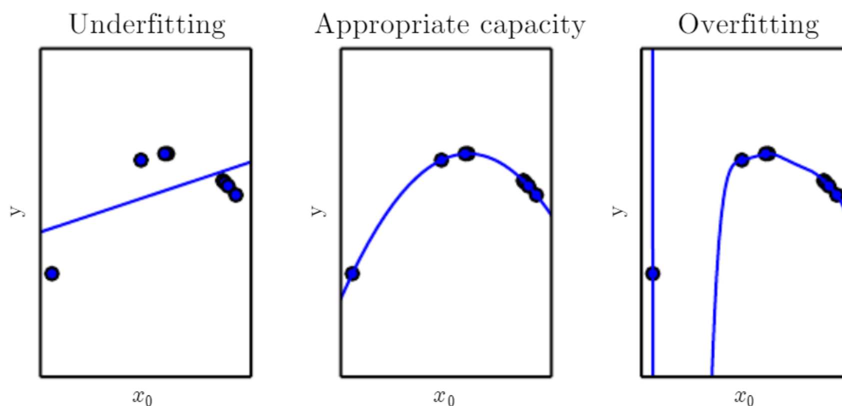


Figure 2.14: Example of a learning algorithm charged with the task of creating a curvature from some given points. On the left it cannot grasp the shape of the function (underfitting). In the middle it adapts the function well to the given points. On the right it assigns the single samples too much weight and estimates the curve incorrectly (overfitting) [taken from 7].

2.3.1 Density Estimation

The purpose of *density estimation* is the construction of an estimated density function $\hat{f}$ using the data set $S := \{\vec{x}_1, \ldots, \vec{x}_M\} \subset \mathbb{R}^d$ of M samples from an unknown distribution with the unknown probability density function f [8]. In general density estimation methods can either be *parametric* or *nonparametric*. Parametric density estimation assumes that there is additional information given about the structure of the distribution besides S, nonparametric does not.
The most popular method of nonparametric density estimation is based on *kernels*. Kernels in one dimension are estimated for every $x_i \in S$ with usually the Gaussian kernel function:

$$K(x) := (2\pi)^{-\frac{1}{2}} \cdot \mathrm{e}^{-\frac{x^2}{2}} \tag{2.42}$$

They are then linearly combined to construct the one-dimensional estimator

$$\hat{f}_{\text{kernel}}(x) := \frac{1}{M} \sum_{i=1}^{M} K\left(\frac{x - x_i}{h}\right) \quad , \tag{2.43}$$

whereas h is the bandwidth or rather the smoothing coefficient. Because the computation of $\hat{f}$ depends on the number of sample points M, kernel based density estimation can become quite expensive for large data sets. To overcome this, the data can be mapped onto a grid, which summarizes multiple sample points $\vec{x}_i \in S$ on single grid points. But this approach suffers from the curse of dimensionality, because the resulting grid is a regular full grid and therefore the number of these grid points increases exponentially. As explained in section 2.2, sparse grids can remedy this problem to some extent.
The basic idea of grid based density estimation is to formulate a highly-overfitted guess f_ϵ of the density function f and then use spline smoothing to turn this into the more representative approximation $\hat{f}$ in a suitable function space V [8] so that the following equation holds:

$$\hat{f}_{\text{grid}}(\vec{x}) = \underset{u \in V}{\operatorname{argmin}} \int_\Omega (u(\vec{x}) - f_\epsilon(\vec{x}))^2 \, d\vec{x} + \lambda \|\mathrm{L}u\|^2_{L^2} \tag{2.44}$$

This equation ensures the closeness of $\hat{f}_{\text{grid}}$ to the initial guess f_ϵ, while also regulating its smoothness with $\|\mathrm{L}u\|^2_{L^2}$, whereas the *regularization parameter* $\lambda > 0$ controls the degree of smoothness [9].
With all test functions $s \in V$ and $f_\epsilon = \frac{1}{M}\sum_{i=1}^{M} \delta_{\vec{x}_i}$, whereas $\delta_{\vec{x}_i}$ is the Dirac delta function centered on sample point $\vec{x}_i$, this leads after some transformations to:

$$\int_\Omega u(\vec{x}) \cdot s(\vec{x}) d\vec{x} + \lambda \int_\Omega \mathrm{L}u(\vec{x}) \cdot \mathrm{L}s(\vec{x}) d\vec{x} = \frac{1}{M} \sum_{i=1}^{M} s(\vec{x}_i) \tag{2.45}$$

Using the sparse grid approach, the finite sub function space $V_n^{(s)} \subset V$ (see section 2.2) or in case of using the combination technique each anisotropic full grid space $V_{\vec{\ell}}^{(c)} \subset V$ (see section 2.2.1) for the sparse grid space $V_{\ell_{\min},\ell_{\max}}^{(c)}$ is considered. The set of all basis functions in $V_n^{(s)}$ or $V_{\vec{\ell}}^{(c)}$ and their corresponding coefficients are defined as:

$$\vec{\Phi} := (\Phi_1, \ldots, \Phi_N) \qquad \text{all } N \text{ basis functions in } V_n^{(s)} \text{ or } V_{\vec{\ell}}^{(c)} \tag{2.46}$$

$$\vec{\alpha} := (\alpha_1, \ldots, \alpha_N) \qquad \text{all } N \text{ corresponding coefficients} \tag{2.47}$$

Note that $\vec{\alpha}$ is still unknown at this point and has to be computed in order to construct the approximated density function:

$$\hat{f}_{\text{sgrid},N}(\vec{x}) := \sum_{i=1}^{N} \alpha_i \cdot \Phi_i(\vec{x}) \tag{2.48}$$

Therefore the following equation must hold for all $\Phi \in \vec{\Phi}$:

$$\int_{\Omega} \hat{f}_{\text{sgrid},N}(\vec{x}) \cdot \Phi(\vec{x}) d\vec{x} + \lambda \int_{\Omega} \mathrm{L}\hat{f}_{\text{sgrid},N}(\vec{x}) \cdot \mathrm{L}\Phi(\vec{x}) d\vec{x} = \frac{1}{M} \sum_{i=1}^{M} \Phi(\vec{x}_i) \tag{2.49}$$

Because $\hat{f}_{\text{sgrid},N}$ is a linear combination of all basis functions $\Phi_i \in \vec{\Phi}$, this can be rewritten as a linear equation system:

$$(\boldsymbol{R} + \lambda \boldsymbol{C})\, \vec{\alpha} = \vec{b} \tag{2.50}$$

$\boldsymbol{R}$, $\boldsymbol{C}$ and $\vec{b}$ are computed with:

$$R_{i,j} = \left(\Phi_i,\, \Phi_j\right)_{L^2} = \int_{\Omega} \Phi_i(\vec{x}) \cdot \Phi_j(\vec{x}) d\vec{x} \tag{2.51}$$

$$C_{i,j} = \left(\mathrm{L}\Phi_i,\, \mathrm{L}\Phi_j\right)_{L^2} = \int_{\Omega} \mathrm{L}\Phi_i(\vec{x}) \cdot \mathrm{L}\Phi_j(\vec{x}) d\vec{x} \tag{2.52}$$

$$b_i = \frac{1}{M} \sum_{j=1}^{M} \Phi_i(\vec{x}_j) \tag{2.53}$$

This linear equation system can then be simplified by replacing the matrix $\boldsymbol{C}$ with the unit matrix $\boldsymbol{I}$ in favor of penalizing non-smooth functions:

$$(\boldsymbol{R} + \lambda \boldsymbol{I})\, \vec{\alpha} = \vec{b} \tag{2.54}$$

For more details, please refer to [8].
Figure 2.15 shows the construction of $\hat{f}$ with a data set and its corresponding sparse grid.

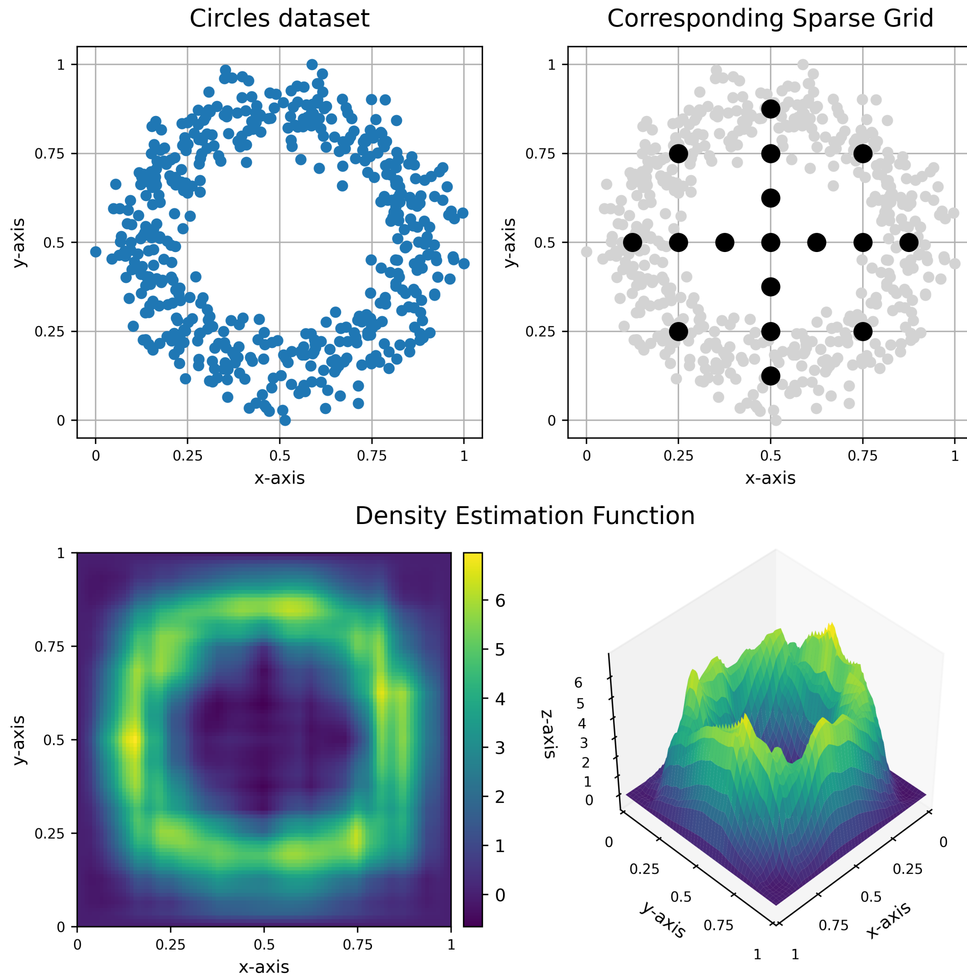


Figure 2.15: Example of the construction of a density function with a sparse grid of level 3. The data set "Circles" (top left) [taken from 10] serves as input S for the density estimation function. Sample points are mapped onto certain sparse grid points (top right). The density function $\hat{f}_{\text{sgrid},17}$ (bottom) is constructed with the constraint that equation 2.49 must hold for the basis functions on every grid point of the sparse grid.

2.3.2 Classification

The goal of classification is to train an algorithm to map some input $\vec{x}$ onto some output class label y within a finite set of classes $y \in [1,\ C]$, whereas C is the number of all possible class assignments [1]. Learning is performed on a given training set $D_{\text{train}} := \{(\vec{x}_i,\ y_i)\}_{i=1}^{N}$ of N total training input points. Each training input $\vec{x}_i$ consists of so called *features* or *attributes*, and is assigned a corresponding training output or *response variable* $y_i \in [1,\ C]$.
There are different types of classification. For $C = 2$, the problem is called *binary classification*, for $C > 2$ the term *multi-class classification* is used. Also class assignments can be combined, e.g. when one input $\vec{x}_i$ is assigned two or more different classes at once, in which case the term is *multi-label classification*.
Learning can be performed with different approaches. Noteworthy are decision trees, neuronal networks, k-nearest neighbors and bayes-classification. Here the function approximation approach is discussed, where the unknown evaluation function g, for which

$$y_i = g(\vec{x}_i), \quad 1 \leq i \leq N \tag{2.55}$$

holds, is approximated as $\hat{g}$ with the help of the training set D_{train}. This approximated function $\hat{g}$ can then map any unknown input $\tilde{\vec{x}}$ onto one class within the given set $\tilde{y} \in [1,\ C]$.

Density-based classification

Applying this to the estimation of a density function, which was discussed in section 2.3.1, the training set of a classification problem can be used to create a method, where density estimation is used to construct a mapping function $\hat{g}$. This is achieved by separating the training set D_{train} into C parts with N_j equally-labeled inputs each (see Figure 2.16):

$$D_{\text{train},j} := \left\{ \left(\vec{x}_i,\ y_j\right) \right\}_{i=1}^{N_j}, \quad 1 \leq j \leq C \tag{2.56}$$

For every subset $D_{\text{train},j}$ a corresponding density function $\hat{f}_j$ is then constructed. To map any unfamiliarly labeled input $\tilde{\vec{x}}$ onto one class, the output $\hat{f}_j(\tilde{\vec{x}})$ for all $j \in [1,\ C]$ is calculated and the class j to the corresponding density function $\hat{f}_j$, which evaluates the highest, is assigned to the given input. The mapping function $\hat{g}$ can therefore be defined as (see Figure 2.17):

$$\hat{g}(\vec{x}) := y_j, \quad \hat{f}_j(\vec{x}) \geq \hat{f}_k(\vec{x})\ \forall k \in [1,\ C] \tag{2.57}$$

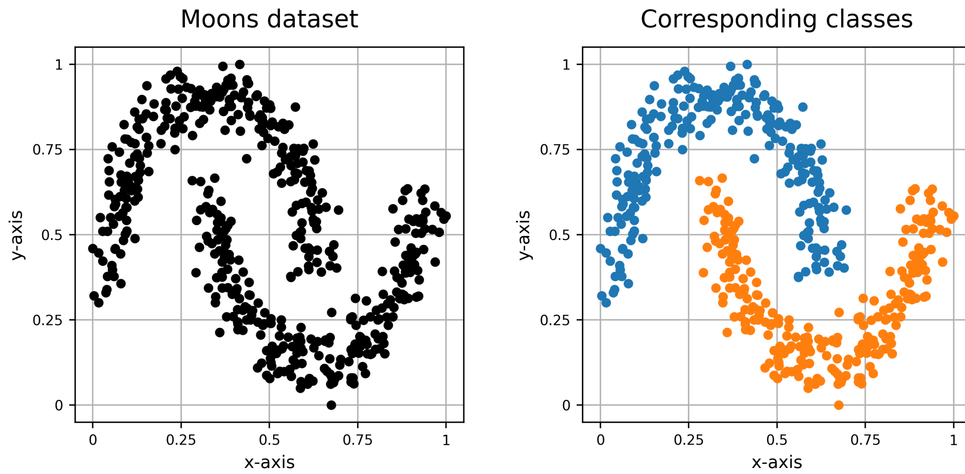


Figure 2.16: Example of a given "Moons" training set set D_{train} (left) [taken from 10], which is split into its sub training sets $D_{\text{train},1}$ and $D_{\text{train},2}$ (right).

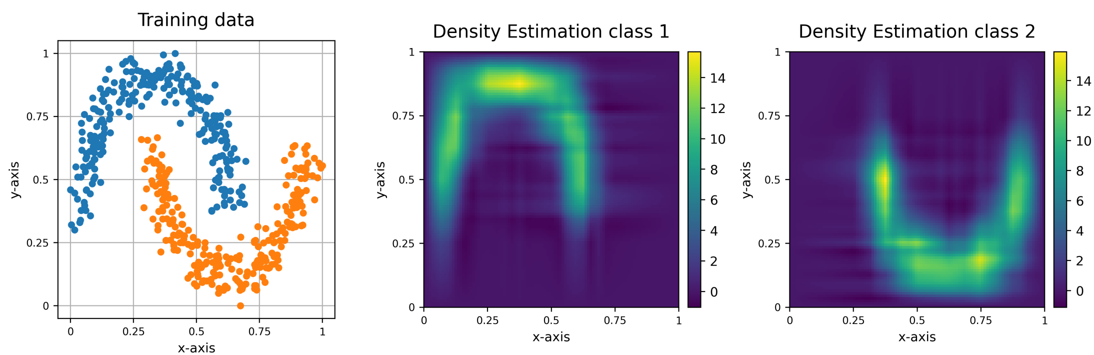


Figure 2.17: Example of performing density estimation on sub training sets $D_{\text{train},1}$ and $D_{\text{train},2}$ of the initial training set D_{train} (see Figure 2.16) independently to construct the mapping function $\hat{g}(\vec{x})$.

2.3.3 Clustering

With *clustering* the goal is to let a computer algorithm analyze a given input set $D := \{\vec{x}_i\}_{i=1}^N$ to find similar properties, create so called *clusters* for the most interesting ones and then categorize every input into one of those clusters. Because this is done completely autonomously by the algorithm without any additional information given other than the input itself, this is also often called *knowledge discovery* [1]. Since the properties of a cluster are unknown before execution of the algorithm, the clustering problem is much less defined than the classification problem and verifying results are much harder. The obvious advantage however is that clustering does not require an expert to supervise the algorithm or manually label the input before execution. Real world machine learning tasks can be rather complex and one or multiple labels are often not enough to represent a certain group of inputs, hence letting the computer handle the labeling results in more representative classes or clusters.
Discovering clusters can be done with various procedures. The most common procedure types are centroid-based or k-means, connectivity-based or hierarchical, distribution-based, grid-based or density-based clustering [11].

Density-based clustering

Taking a closer look at density-based clustering, the basic idea is to select all regions within the input set with a relatively high density of input points compared to the surrounding areas, whereas each of those regions represents one cluster. [12].
The first step of density-based clustering is to build the estimated density function $\hat{f}$ for the input set D, which can be based on a sparse grid (see equation 2.48). Defining then a *density threshold* t_d, all input points $\vec{x}_i \in D$ with an estimated density above this threshold are filtered into one region $R(t_d)$:

$$R(t_d) := \left\{ \vec{x}_i \in D \mid \hat{f}(\vec{x}) \geq t_d \right\} \tag{2.58}$$

This region can consist of one or more connected components, indicated by *valleys* between input points in $R(t_d)$ (see Figure 2.18).
However, determining how many connected components there are or which input points can be assigned to which connected component can be quite difficult to implement. The algorithm created utilizes the *k-nearest-neighbor-graph* approach to connect every input point $\vec{x}_i$ to its k-nearest neighbors and uses the estimated density function $\hat{f}$ to determine, which graph edges should be cut to construct the connected components. Selecting those edges can be done in various ways, e.g. all edges overlapping with previously described valleys could be omitted. An intuitive approach is to estimate the density for an edge and omit it, if its density is below a certain *cutting threshold* t_c.

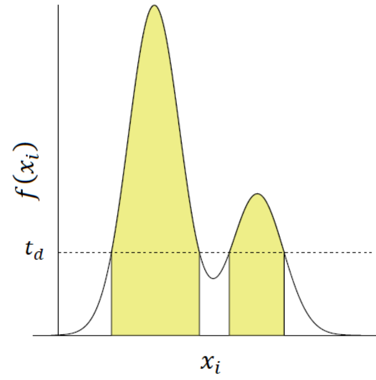


Figure 2.18: Example of the density distribution of a one-dimensional input set D. All input points x_i with $\hat{f}(x_i) \geq t_d$ (marked in yellow) are filtered into $R(t_d)$, which consists of two connected components. Between those there is a valley of points $\tilde{x}_i \notin R(t_d)$ [taken from 12].

The most simple, but not necessarily most accurate way to accomplish this is to determine the midpoint $\vec{x}_{\text{mid}(i,j)}$ of two points $\vec{x}_i$ and $\vec{x}_j$ connected by an edge and computing its density:

$$\hat{f}_{\text{mid}(i,j)}(\vec{x}_i,\ \vec{x}_j) := \hat{f}\left(\frac{1}{2} \cdot (\vec{x}_i + \vec{x}_j)\right) \tag{2.59}$$

Omitting every edge $e_{i,j}$ with its density $\hat{f}_{\text{mid}(i,j)} \leq t_c$ results in $n \in \mathbb{N}_0$ independent graphs, which each is a detected cluster. All points which are left without any connecting edges are considered *noise* and are either removed or connected to their nearest cluster. Note that if all edges of the k-nearest-neighbor graph were omitted and $n = 0$, all input points $\vec{x} \in D$ are considered noise. In this case the algorithm can either fail, consider each point its own cluster or simply connect all points to a single cluster.
After all clusters are determined, depth-first-search can be used to assign every input point its corresponding cluster.
Figure 2.19 shows the step-by-step procedure of the described clustering algorithm.

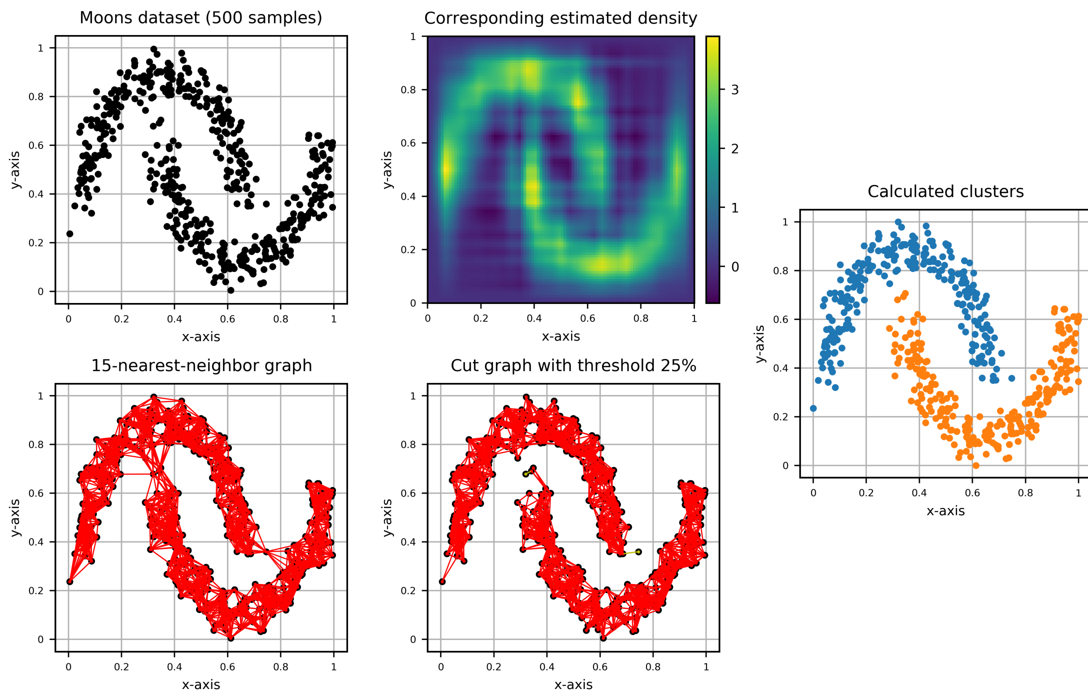


Figure 2.19: Example of the procedure of clustering a "Moons" data set [taken from 10] (top left) with a density estimation based on a combination-technique sparse grid with $\ell_{\min} = 1$ and $\ell_{\max} = 6$ (top mid). With this the 15-nearest-neighbor graph is build (bottom left) and then cut into two connected components with a cutting threshold of $t_c = 0.25$ (bottom mid). Note that there was detected some noise (marked in green), which was added to its nearest corresponding cluster. After that each data point is assigned one of those two clusters with recursive depth-first-search (mid right).

3 Implementation

In the scope of this thesis, a density-based classification and clustering algorithm was implemented into the python framework *Sparse Grid Spatially Adaptive Combination Environment*, short *sparseSpACE*[1], which was created by Michael Obersteiner. The in 2020 by Lukas Schulte integrated *Density Estimation* functionality [13] was used to estimate the density functions for both machine learning algorithms.
This chapter first provides a short introduction to the existing implementations and functionalities of the *sparseSpACE* framework in section 3.1, including the implementation of the aforementioned density estimation functionality, before explaining the `DEMachineLearning` wrapper in section 3.2, which combines density based classification and clustering algorithms in one python module.

3.1 The sparseSpACE-framework

Originally created to numerically integrate high dimensional functions with the spatially adaptive combination technique, the *sparseSpACE* framework nowadays also supports the implementation of arbitrary grid based operations with various alterations of the combination technique. Every in the framework implemented operation is a subclass of the `GridOperation` superclass and is to be found in the module with the same name. Solving an operation is done with the help of one of the available combination technique classes, e.g. for the standard combination technique the `StandardCombi` class is used. Also the type of grid used for any computation can be specified by one of the available grid subclasses located in the `Grid` module, which all provide some utility for the underlying operations, e.g. returning the number of total grid points for a given discretization level.
The `DensityEstimation` class, derived from `GridOperation`, implements the estimation of a density function as explained in chapter 2, section 2.3.1 on `TrapezoidalGrids` with the help of the `StandardCombi` class. For a more detailed explanation on this, please refer to [13].

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

3.2 The DEMachineLearning wrapper

Because the underlying density estimation functionality for both in chapter 2, section 2.3.2 and 2.3.3 discussed machine learning algorithms is already integrated into the *sparseSpACE* framework, it makes sense to design a *wrapper* as an interface for the classification and clustering algorithm. This wrapper was realized as the `DEMachineLearning` module, which contains the three classes `DataSet`, `Classification` and `Clustering`.
As machine learning is usually performed on some data set D, it can be helpful to have a certain utility class available, which provides various functionalities for data sets, e.g. separating them based on labels for classification tasks or scale them to a certain range. For this purpose the `DataSet` class was implemented to facilitate the implementation of the `Classification` and `Clustering` class, presented in section 3.2.1 and section 3.2.2. A `DataSet` object mainly consists of a tuple with two entries, whereas the first one contains the actual data and the second one the corresponding label to every sample point.
For a tutorial on how to use the `DEMachineLearning` interface, please refer to the corresponding IPython-Notebook-Tutorial[2] in the *sparseSpACE* framework.

3.2.1 The Classification class

The basic functionality of the `Classification` class is to learn the distribution of some classes in an initial data set D and then label samples of arbitrary input data sets based on the performed learning. This process can be split into four separate key steps:

- Preprocessing of the initial data set D into D_{train} and possibly D_{test}.
- Learning the structure of D_{train} with density estimation.
- Mapping classes on D_{test}, if specified before, or on arbitrary new data sets based on the previously performed learning.
- Evaluating the received results.

The next few sections describe this pipeline in more detail, while simultaneously giving a brief overview of the program structure of the `Classification`-implementation with some python pseudo-code.

[2] https://github.com/obersteiner/sparseSpACE/tree/master/ipynb

Classification: Preprocessing

When initializing a `Classification` object, preprocessing is done automatically. The constructor accepts one required parameter, a `DataSet` object as initial data set D, and four optional parameters, whereas the *splitting percentage* is the one most worth mentioning. This parameter determines, which percentage of D should be used as the training set D_{train} and consequently which remaining part should be the testing set D_{test} for evaluation purposes. It defaults to 1 or 100%, meaning an empty training set. Before splitting, D is scaled to the range $[0, 1]$ for the `DensityEstimation` algorithm, which is part of the subsequent learning process. The reason for this is the underlying sparse grid and its basis functions (see chapter 2). Furthermore not only the training subset $D_{\text{train}} \subseteq D$ is extracted, it is also further divided into C sub training data sets $D_{\text{train},j}$ with $j \in [1, C]$, whereas C is the number of total classes in D_{train}, preparing the algorithm for the upcoming learning process (see Figure 3.1).
With two optional constructor parameters can also be specified, whether an even distribution among classes in the training set should be forced or a random distribution of samples in D before splitting should be ensured.

```python
def preprocessing(initial_dataset, splitting_percentage):
    initial_dataset.scale_range([0, 1])
    length_data = len(initial_dataset)
    data_train = initial_dataset[:splitting_percentage * length_data]
    data_test = initial_dataset[splitting_percentage * length_data:]
    list_train_subsets = data_train.divide_into_subsets_by_class()
```

Figure 3.1: Python pseudo-code of the basic implementation of the preprocessing step in the `Classification` class.

Classification: Learning process

After a `Classification` object is created and consequently all training subsets $D_{\text{train},j}$ are determined, the operation `perform_classification` (see Figure 3.2) can be called on mentioned object to start the learning process. Here for every training subset or rather individual class one density estimation is performed to create `classificators`, which are stored within a protected list in the `Classification` object. Parameters for the `perform_classification` method include all necessary parameters to create a `DensityEstimation` object, i.a. the minimal and maximal level of the underlying sparse grid or the regularization parameter λ. The learning process can only be executed once per object.

```
def perform_classification(min_level, max_level, lambd):
    list_classificators = []
    for subset in list_train_subsets:
        list_classificators += subset.density_estimation(min_level,
                                                          max_level,
                                                          lambd)
    # testing data is evaluated directly if present
    if data_test is not empty:
        data_test_computed = classify_new_data(data_test)
        evaluate_classification(data_test, data_test_computed)
```

Figure 3.2: Python pseudo-code of the basic implementation of the learning step in the `Classification` class.

Classification: Labeling process

Once the learning process on a `Classification` object is completed, either all samples of D_{test} are directly labeled, if it was created in the preprocessing step (see Figure 3.2, bottom), or a completely new `DataSet` object $\tilde{D}$ can be classified based on the performed learning (see Figure 3.3). In both instances each `classificator` created in the learning process is evaluated for all samples of either D_{test} or $\tilde{D}$ and the corresponding class of one `classificator`, which evaluated the highest for a given sample, is assigned to that sample.

```
def classify_new_data(new_dataset):
    computed_classes = []
    for sample in new_dataset:
        list_densities_for_sample = []
        for classificator in list_classificators:
            list_densities_for_sample += classificator(sample)
        # argmax(x) returns the position for the max value in list x
        computed_classes += argmax(list_densities_for_sample)
    return new_dataset.add_classes(computed_classes)
```

Figure 3.3: Python pseudo-code of the basic implementation of the labeling step in the `Classification` class.

Classification: Evaluation

To evaluate the computed labeling of the testing or any new input data set, a list of all consecutive computed classes is compared component-wise with the classes of the used data set. This results in a list of equal length with the entry 0 for every correct and 1 for every incorrect labeling. Summing all values within this list up results in the total number of incorrectly classified samples (see Figure 3.4). Besides the evaluation of correct or incorrect classifications, the time used for learning and labeling is measured and can be returned in the evaluation step.

```
def evaluate_classification(original_dataset, computed_dataset):
    list_results_labeling = []
    original = original_dataset.get_classes()
    computed = computed_dataset.get_classes()
    for class_orig, class_comp in zip(original, computed):
        list_results_labeling += class_orig != class_comp
    return sum(list_results_labeling)
```

Figure 3.4: Python pseudo-code of the basic implementation of the evaluation step in the `Classification` class.

3.2.2 The Clustering class

The purpose of the `Clustering` class is to label each sample of an input data set D, which may or may not already have assigned labels to its samples. The resulting classified data set can then e.g. be used for a classification task. This clustering process can be divided into six key steps:

- Preprocessing of the initial data set D.
- Learning the structure of D with density estimation.
- Building the full nearest-neighbor-graph.
- Cutting aforementioned graph based on the learned density.
- Determining the connected components or clusters.
- Evaluating the received results.

Again the next few sections give further insight into this pipeline, while also presenting the structure of the basic `Clustering`-class-implementation with python pseudo-code.

Clustering: Preprocessing

The `Clustering` class has a similar composition like the `Classification` class, which is why preprocesing also happens right after initializing a `Clustering` object. Because there is not much information available to a clustering algorithm other than the input data set D itself (see chapter 2, section 2.3.3), not much preprocessing has to be done except of again scaling D to the range $[0, 1]$ (see Figure 3.5). Nonetheless beside the required parameter of the input set, two important other parameters can be specified optionally; the number of nearest neighbors k, which defaults to 5 and the edge-cutting-threshold t_c, which defaults to 0.25 or 25%. Both are used in the subsequent processes of building and cutting the nearest-neighbor-graph.

```
def preprocessing(initial_dataset, nearest_neighbors, threshold):
    initial_dataset.scale_range([0, 1])
    # the two parameters 'nearest_neighbors' and 'threshold', which are
    # here passed to the constructor are needed for later steps in the
    # clustering algorithm
```

Figure 3.5: Python pseudo-code of the basic implementation of the preprocessing step in the `Clustering` class.

Clustering: Learning process

Like for the `Classification` class, once a `Clustering` object is initialized and preprocessing is completed, the `perform_clustering` operation (see Figure 3.6) can be called on said object to initiate the learning process. A `DensityEstimation` operation is performed on the whole initial data set D to receive a `clusterinator` or density function based on D, which is used for determining the edges to cut out of the full nearest-neighbor-graph after it is build. Parameters for the `perform_clustering` operations again mainly consist of the parameters used for initializing the `DensityEstimation` object.

```
def perform_clustering(min_level, max_level, lambd):
    clusterinator = initial_dataset.density_estimation(min_level,
                                                       max_level,
                                                       lambd)
```

Figure 3.6: Python pseudo-code of the basic implementation of the preprocessing step in the `Clustering` class.

Clustering: Building the nearest-neighbor-graph

The full nearest neighbor graph is build with the `sklearn.neighbors` module [10], which fits one data set $\tilde{D}_1$ onto another one $\tilde{D}_2$ and detects all k `nearest_neighbors` (constructor parameter, see Figure 3.5) in $\tilde{D}_2$ for every sample in $\tilde{D}_1$. Here this is utilized by fitting the initial data set D onto itself, hence finding all k nearest neighbors for every sample within D. This however results in every sample having itself as one detected nearest neighbor, which is why actually $k+1$ nearest neighbors have to be detected. After that all redundant edges are removed and the resulting list of all connecting edges is stored into a protected list within the `Clustering` object (see Figure 3.7).

```
def build_nearest_neighbor_graph():
    nearest_neighbors += 1
    # 'neighbors' contains all edges, also those with redundancy
    neighbors = initial_dataset.fit(initial_dataset, nearest_neighbors)
    list_all_edges = list(neighbors.remove_redundant_edges())
```

Figure 3.7: Python pseudo-code of the basic implementation of the nearest-neighbor-graph-building step in the `Clustering` class.

Clustering: Cutting the nearest-neighbor-graph

After the nearest-neighbor-graph is built, the previously estimated density function (see Figure 3.6) and the cutting `threshold` (constructor parameter, see Figure 3.5) are used to cut all edges below said threshold, constructing a reduced version of the initial nearest-neighbor-graph. There are two separate sub-steps to this process; finding clusters in places with high density and assigning evaluated noise to those clusters. See Figure 3.8 for the corresponding python pseudo-code.
For the first sub step the midpoint $\vec{x}_{\text{mid}(i,j)}$ of each edge in the nearest-neighbor-graph is detected and its density value $\hat{f}_{\text{mid}(i,j)}$ (see equation 2.59) is calculated. With the `DensityEstimation` variable `extrema`, which stores the minimal and maximal value of a density function, the total density percentage of $\hat{f}_{\text{mid}(i,j)}$ can be computed. If this percentage is below the `threshold` parameter, its corresponding edge is removed from the nearest-neighbor-graph.
This removal of edges possibly results in some noise, i.e. samples, whose edges were all omitted. To assign each noise sample exactly one possible cluster in sub step two, the set of noise samples D_{noise} is fit onto the data set, whose samples each have at least one connecting edge, $D_{\text{connected}}$, to find exactly one nearest-neighbor-edge for each noise

sample. Those edges are then added to the existing ones to build the final connected components.

```
def cut_nearest_neighbor_graph():
    # remove all edges below the threshold
    for edge in list_all_edges:
        midpoint = (edge.get_point(0) + edge.get_point(1)) / 2
        density_midpoint = clusterinator(midpoint)
        perc_mid = density_midpoint / clusterinator.extrema.max_value
        if perc_mid < threshold:
            list_all_edges.remove(edge)

    # 'list_all_edges' now contains all edges above the threshold
    # noise edges need to be added
    connected_samples = list_all_edges.get_contained_samples()
    noise_samples = initial_dataset.remove(connected_samples)
    # exactly 1 connected for each noise sample has to be found
    noise_edges = list(noise_samples.fit(connected_samples, 1))
    list_all_edges += noise_edges
```

Figure 3.8: Python pseudo-code of the basic implementation of the nearest-neighbor-graph-cutting step in the `Clustering` class. Divided into the two sub steps "finding clusters" (top) and "appending noise" (bottom).

Clustering: Finding the connected components

With the nearest-neighbor-graph built, cut according to the threshold, and all noise edges added to the graph, all connected components already exist within the graph structure; the algorithm only has to detect them. This is done by recursive depth-first-search (see Figure 3.9), which starts at one sample in D and visits new samples by traversing the given edges in `list_all_edges` (see Figure 3.8). Every sample visited that way is marked as "visited" and once no samples can be reached through edges anymore, another "unvisited" sample in D is selected for the same process. Repeating this until no more samples in D are unvisited results in a list of subsets of D, which form the connected components or clusters. Each component's samples are then labeled with some label $y \in [0,\ C-1]$, whereas C is the number of detected clusters.

```
def find_connected_components():
    unvisited = initial_dataset.get_all_samples()
    connected_components = []
    while unvisited is not empty:
        start_sample = unvisited[0]
        component = depth_first_search(unvisited, start_sample, [])
        connected_components += [component]
    for y, component in enumerate(connected_components):
        for sample in component:
            sample.set_label(y)

def depth_first_search(unvisited, current_sample, component):
    component += [current_sample]
    unvisited.remove(current_sample)
    connecting_edges = list_all_edges.filter_connecting(current_sample)
    for edge in connecting_edges:
        connect_sample = edge.connecting(current_sample)
        if connected_sample is in unvisited:
            c = depth_first_search(unvisited, connect_sample, component)
            component += [c]
    return component
```

Figure 3.9: Python pseudo-code of the basic implementation of the detection step for the connected components in the `Clustering` class.

Clustering: Evaluation

When evaluating the results of the `Clustering` class, the assumption that the initial data set possesses labels to begin with is made. However with the evaluation two problems arise: The number of detected clusters can differ from the number of original ones and also the labeling of detected and original clusters may not necessarily be consistent, e.g. a correctly detected cluster can be labeled with y_1, when its original label was y_2.
The solution here is to determine the difference $\Delta = C_{\text{computed}} - C_{\text{original}}$, whereas C_{computed} is the number of computed and C_{original} the number of original clusters. When $\Delta > 0$, the Δ smallest clusters in `connected_components` (see Figure 3.9) only contain incorrectly mapped samples, which are omitted directly from the evaluation. The cases $\Delta = 0$ and $\Delta < 1$ don't make a difference for the rest of the evaluation, which iterates over all sets of indices, each for one label in `connected_components`, and filters all samples in the initial data set at given index set for the largest original cluster. The indices of all samples in this cluster are then the indices of the correctly labeled samples in `connected_components` for the given label and the rest is added to the samples, which were labeled incorrectly.

```
def evaluate_clustering():
    number_original_clusters = initial_dataset.get_number_labels()
    number_computed_clusters = len(connected_components)
    difference = number_computed_clusters - number_original_clusters
    number_wrong = 0
    if difference > 0:
        # all 'difference' smallest clusters are definitely wrong
        wrong_clus = connected_components.smallest_clusters(difference)
        number_wrong += wrong_clus.get_total_number_samples()
        connected_components.remove(wrong_clus)
        initial_dataset.remove(wrong_clus)
    for label in connected_components.get_labels():
        indices_label = connected_components.get_indices(label)
        compare_samples = initial_dataset[indices_label]
        number_correct = compare_samples.get_largest_cluster()
        number_wrong += len(compare_samples) - number_correct
    return number_wrong
```

Figure 3.10: Python pseudo-code of the basic implementation of the evaluation step in the `Clustering` class.

4 Results

In this chapter various results of the implementation discussed in chapter 3 will be presented and analyzed. All data sets used for analyzation are taken from the *scikit-learn API reference*[1] and the underlying combination-technique implementation is part of the *sparseSpACE*[2] framework designed by Michael Obersteiner.
Results are organized into section 4.1 for classification results and section 4.2 for clustering results. At the beginning of each section some data sets with corresponding performance measures, which compare full with sparse grids for the respective machine learning method, are presented. Then for classification various combinations of ℓ_{min} and ℓ_{max} for combination-technique sparse grids in low and high dimensions are analyzed. The remaining results for each section are of more specific nature, i.e. confusion matrices and varying learning percentages for classification and analyzation of t_c and k with respect to accuracy and complexity for clustering.
Data sets taken from scikit-learn for analyzation are either directly loaded or fetched from the API and therefore static in nature or randomly generated with artificial data generators. For the purpose of this chapter five generated data set types were selected for analyzation:

- The "Circles" data set; a well known 2D data set with two concentric circles and two corresponding classes. Suited mainly for classification, because the circles are too close to each other to perform well on clustering (see section 4.2).
- The "Moons" data set; an also well known 2D data set with two interleaving half circles and two corresponding classes. Suited for both classification and clustering.
- The "Classification" data set; varying in number of classes and dimensions, this set is as the name suggests mainly used for classification purposes.
- The "Blobs" data set; also varying in number of classes and dimensions, this set is mainly used for clustering purposes, but can also serve as a more trivial classification problem.

[1] `https://scikit-learn.org/stable/modules/classes.html`
[2] `https://github.com/obersteiner/sparseSpACE`

- The "Gaussian Quantiles" data set; again varying in number of classes and dimensions, this set is only really useful as a classification problem. It stands out because of its relatively bad approximation through sparse grids due to its shape (see Figure 4.5).

Since for most test cases one performance measurement was computation time, note that the computation of all results presented here were done on a home computer[3]. Performing the same tests on e.g. a high performance data center like the LRZ[4] will probably result in lower computation times.

4.1 Classification

Classification on some data set D is usually evaluated by separating D into a *learning* and *testing* data subset. As explained in chapter 3, the percentage of D chosen for training the algorithm can be manually adjusted with the *sparseSpACE* classification implementation. This section shows and explains results of all previously stated classification problems in terms of how input parameters like discretization levels ℓ and ℓ_{max}, the number of classes, dimensions and the training subset percentage affect output accuracy, measured in the percentage of correctly mapped training points, and complexity, measured in time for the algorithm to run.

4.1.1 Full vs. Sparse grids

Starting with the comparison of how well sparse grids perform on the classification tasks in comparison to full grids, some generated two- and three-dimensional data sets (see Figure 4.1, Figure 4.2, Figure 4.3 and Figure 4.4) are used to compare accuracy and complexity based on (maximal) discretization levels of the grids used for classification on those sets. For every data set, the number of samples amounts to 5000, the training percentage was set to 80% and $\ell_{\text{min}} = 1$ for every corresponding sparse grid.

[3]Quad-Core CPU @ 4.00GHz, 16 GB DDR4 RAM @ 2666MHz
[4]`https://www.lrz.de/`

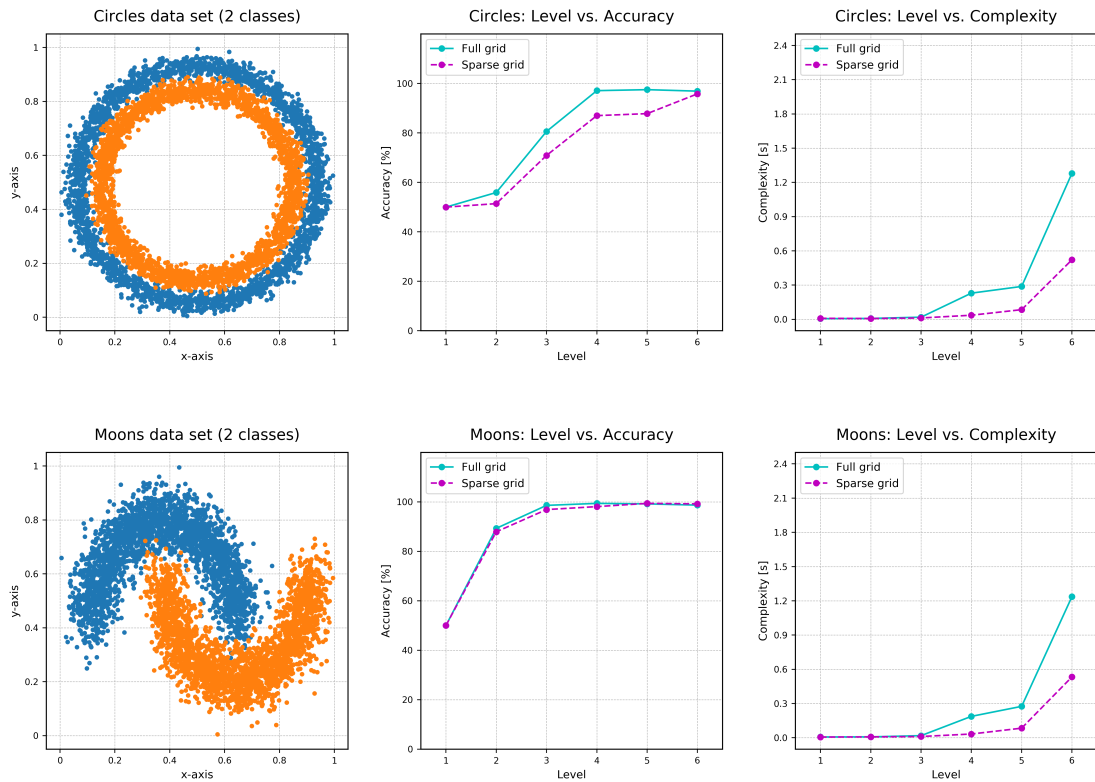


Figure 4.1: Accuracy and complexity based on discretization levels ℓ and $\ell_{\max}$ between 1 and 6 for the classification tasks of data sets "Cirlces" with 5% noise (top) and "Moons" with 15% noise (bottom).

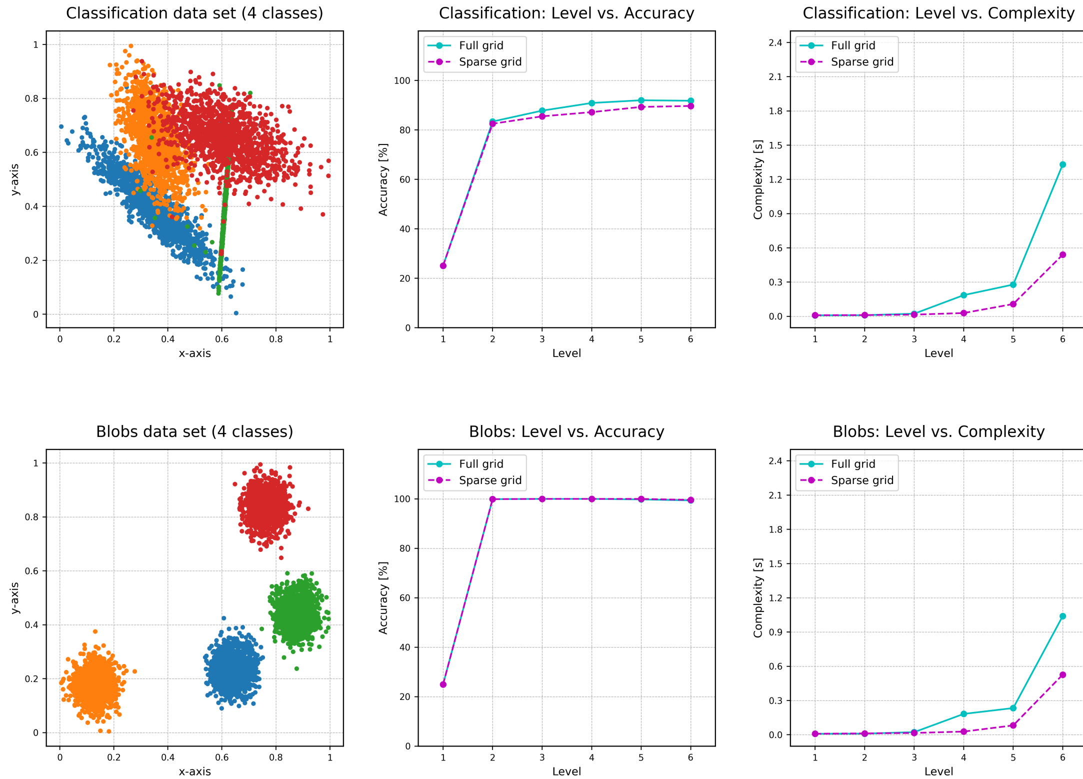


Figure 4.2: Accuracy and complexity based on discretization levels ℓ and $\ell_{\max}$ between 1 and 6 for the classification tasks of data sets "Classification" (top) and "Blobs" (bottom) with 4 classes each.

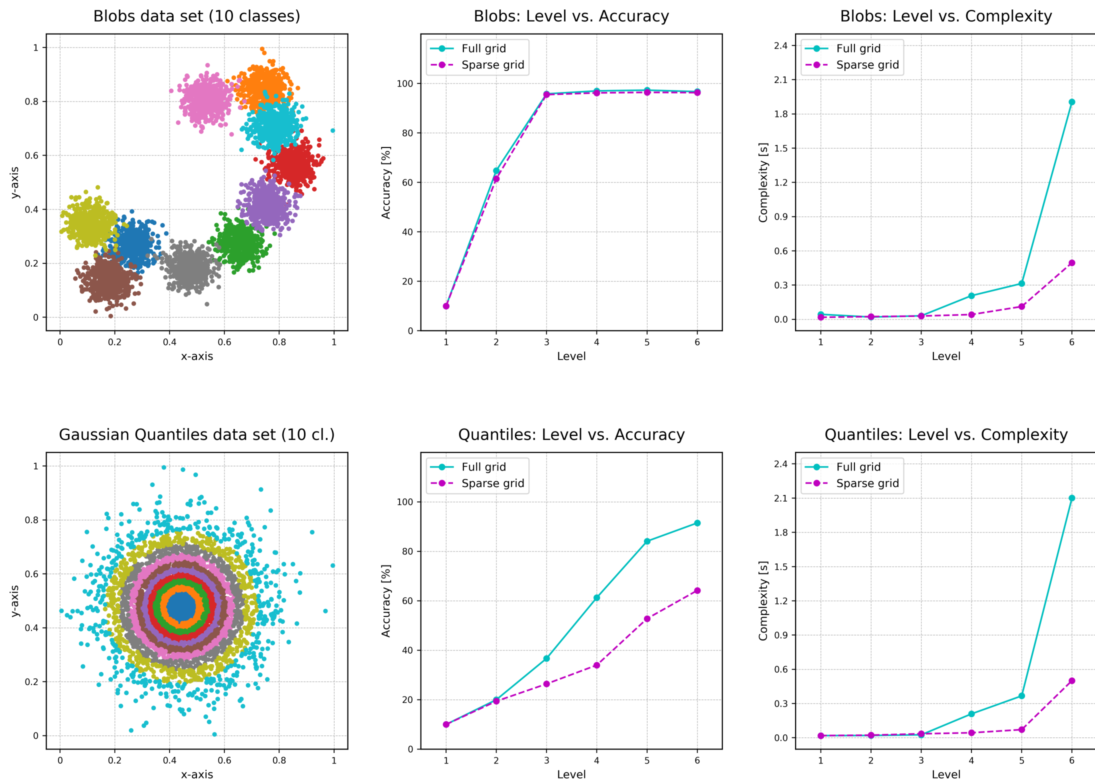


Figure 4.3: Accuracy and complexity based on discretization levels ℓ and $\ell_{\max}$ between 1 and 6 for the classification tasks of data sets "Blobs" (top) and "Gaussian Quantiles" (bottom) with 10 classes each.

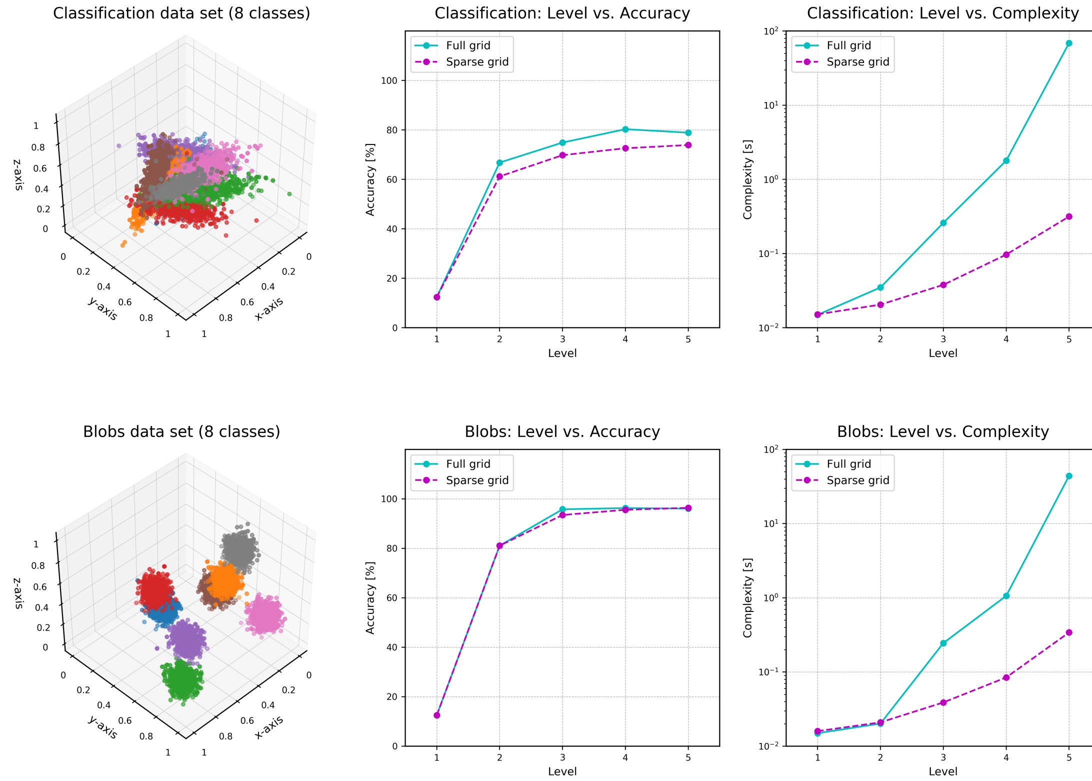


Figure 4.4: Accuracy and complexity based on discretization levels ℓ and $\ell_{\max}$ between 1 and 5 for the classification tasks of the three-dimensional data sets "Classification" (top) and "Blobs" (bottom) with 8 classes each.

General observations

The accuracy of every data set at level 1 for both full and sparse grid seems to be $\frac{100}{C}\%$, whereas C is the number of classes. The reason for this is that at such a low level, every training point is mapped onto a single class. Because the classes are evenly distributed among all samples for the shown data sets, at least one class is always mapped on correctly. Also interesting to note is that the sparse- and full-grid-accuracy of all examples seem to hit a maximal value at some level, which is most of the time slightly below 100% because of the integrated noise.
In line with the number of grid points (see equation 2.33 and 2.40) does the complexity of each full grid computation exceed the one of its corresponding sparse grid. Even for all only two-dimensional examples, the complexity for full grids in comparison to sparse grids clearly shows here:

$$\mathcal{O}\left(2^{2\ell}\right) \qquad \text{complexity of a 2D full grid} \tag{4.1}$$

$$\mathcal{O}\left(2^{\ell_{\max}} \cdot \ell_{\max}\right) \qquad \text{complexity of a 2D sparse grid} \tag{4.2}$$

Even more so for the three-dimensional examples in Figure 4.4:

$$\mathcal{O}\left(2^{3\ell}\right) \qquad \text{complexity of a 3D full grid} \tag{4.3}$$

$$\mathcal{O}\left(2^{\ell_{\max}} \cdot {\ell_{\max}}^2\right) \qquad \text{complexity of a 3D sparse grid} \tag{4.4}$$

All while the accuracy curves for each sparse and full grid computation seem to accept relatively low discrepancies, except for the "Circles", 3D-"Classification" and especially "Gaussian Quantiles" data sets. Two main reasons for this come to mind: Adaptively unrefined, borderless sparse grids approximate sample points in the corners worse than any other ones. When a data set has relatively many corner points, a full grid pulls ahead of a sparse grid in terms of accuracy. For the same reason circular shaped clusters of data or whole data sets also are suboptimal for computations on sparse grids, because they will be interpreted as rectangular shaped data on lower levels (see Figure 4.5). This becomes even more clear when looking at the way a combination-technique-sparse-grid is constructed (see Figure 4.6).
Easy to miss but interesting nonetheless is the fact that for some examples the computation time of the same set differs for full and sparse grid computations at level 1. This should not happen, since at this level both grids have the exact same amount of grid points. A possible explanation for this could be fluctuations in the corresponding computation medium, e.g. some other processes the CPU has run at the same time.

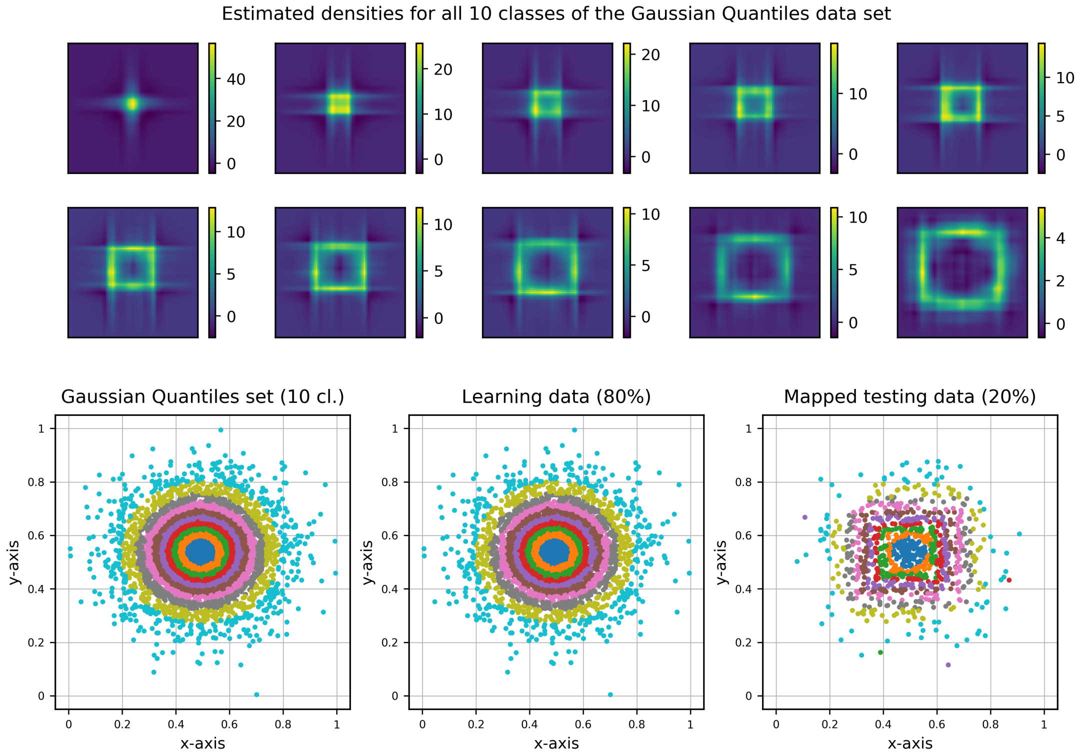


Figure 4.5: Example of a classification task with $\ell_{\text{min}} = 1$ and $\ell_{\text{max}} = 6$ for a two-dimensional "Gaussian Quantiles" data set with 10 classes and 80% learning data. Because of the sparse grid construction of all 10 density functions (top), circles of testing samples appear to be mapped with a rectangular shape (bottom right).

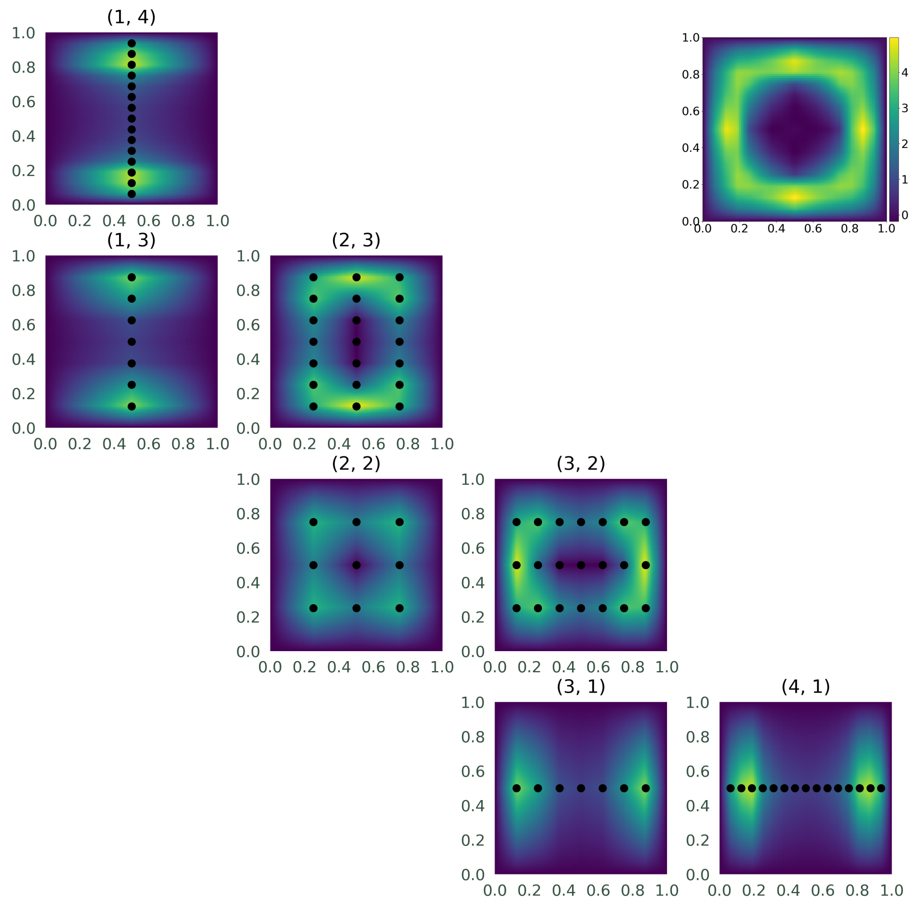


Figure 4.6: Example of the step-by-step construction of the density function for some "Circles" data set with the combination technique with $\ell_{\text{min}} = 1$ and $\ell_{\text{max}} = 4$. The result can be seen in the top right corner.

$\ell_{\min} \setminus \ell_{\max}$	1	2	3	4	5	6
1	50.0%	50.5%	72.0%	87.8%	88.9%	95.4%
2		53.4%	83.6%	94.0%	96.6%	97.0%
3			81.8%	96.7%	97.1%	97.1%
4				97.2%	97.4%	97.6%
5					97.4%	97.6%
6						97.5%
$\ell_{\min} \setminus \ell_{\max}$	1	2	3	4	5	6
1	0.007s	0.009s	0.017s	0.043s	0.095s	0.761s
2		0.025s	0.024s	0.063s	0.710s	2.225s
3			0.021s	0.682s	1.570s	2.180s
4				0.342s	1.016s	1.830s
5					0.382s	1.627s
6						1.586s

Table 4.1: Accuracy (top) and complexity (bottom) of the "Circles" data set (see Figure 4.1) for the two input parameters minimal level $\ell_{\min}$ and maximal level $\ell_{\max}$.

4.1.2 Minimal vs. Maximal Level

Since a combination-technique sparse grid is defined trough its minimal and maximal level, a varying proportion of those two parameters can be interesting to analyze. In general the closer $\ell_{\min}$ and $\ell_{\max}$ are, the more "full-grid-like" a sparse grid becomes, until the case for $\ell_{\min} = \ell_{\max}$ results in a regular full grid. In Table 4.1 the accuracy and complexity based on those two input parameters for the "Circles" data set from Figure 4.1 are shown. As expected do higher input level parameters result in a rise of accuracy. Interesting however is the fact, that for all instances of $1 < \ell_{\min} < \ell_{\max}$ the complexity exceeds both the sparse grid with a minimal level of 1 and more importantly the regular full grid. This has to do with the constraints for which the anisotropic full grids are chosen. Looking at this specific example for a two-dimensional combination-technique sparse grid, all anisotropic full grids are added when $|\vec{\ell}|_1 = \ell_{\max} + \ell_{\min}$ and subtracted when $|\vec{\ell}|_1 = \ell_{\max} + \ell_{\min} - 1$ (see equation 2.38). Whenever $\ell_{\min}$ nears $\ell_{\max}$, less anisotropic full grids have to be combined, but in return those are more complex. E.g. a single regular full with $\ell_{\min} = \ell_{\max}$ seems to be less complex than a combination of some "almost-full-grids" with $\ell_{\min} = \ell_{\max} - 1$.

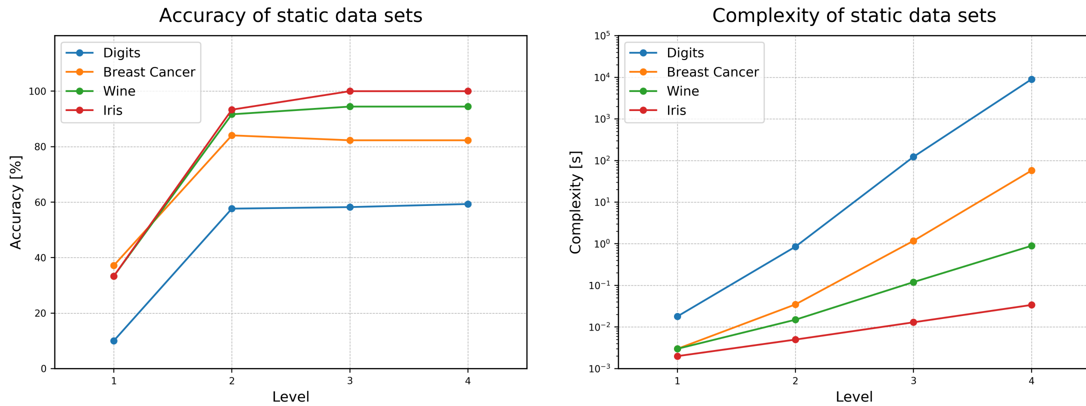


Figure 4.7: Accuracy (left) and complexity (right) of four from scikit-learn loaded static data sets. Accuracy is measured by the percentage of correct mappings of the testing data. Complexity is measured by the computation time of the classification task and displayed on a logarithmic scale.

4.1.3 Static data sets

Besides generated data sets scikit-learn also provides static ones specifically for classification tasks. Here a subset of five loaded data sets was selected to show comparable meta results for the *sparseSpACE* implementation of classification and similarities of complexity for different dimensions:

- The "Digits" data set; 64-dimensional with 10 classes and 1797 samples.
- The "Breast cancer" data set; 30-dimensional with 2 classes and 569 samples.
- The "Wine" data set; 13-dimensional with 3 classes and 178 samples.
- The "Iris" data set; 4-dimensional with 3 classes and 150 samples.

Figure 4.7 shows the accuracy and complexity of all four sets with input parameter $1 \leq \ell_{\text{max}} \leq 4$. As in section 4.1.1 every accuracy seems to hit a certain maximal value or *ceiling* at some level, which does not significantly increase with higher levels. Also as observed previously in section 4.1.1, the accuracy at level 1 for every data set seems to be $\frac{100}{C}$% with the number of classes C. An exception to this is the "Breast Cancer" set, because unlike all other sets, both its classes are not evenly distributed among all samples. Their sampling ratio is approximately 60/40, which explains the accuracy of slightly below 40% for the "Breast Cancer" set at level 1.

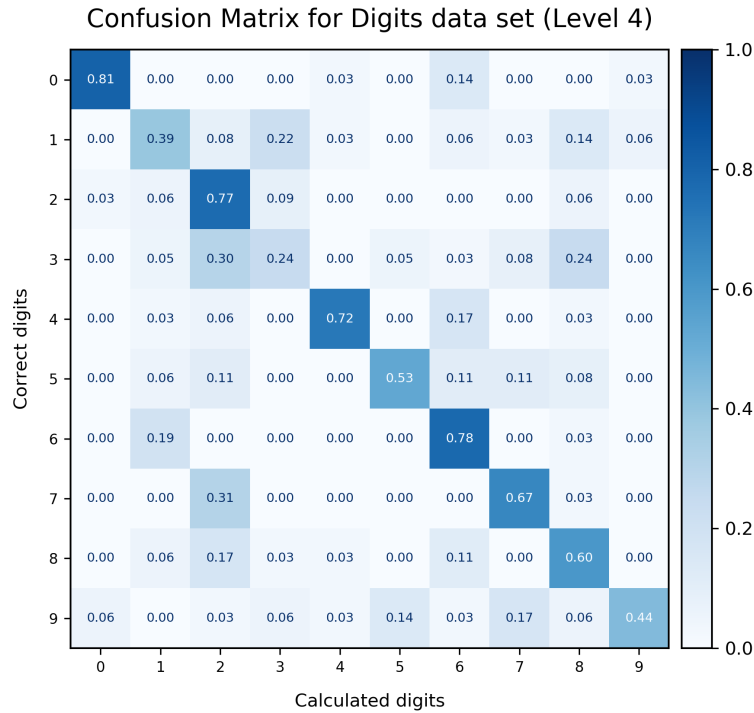


Figure 4.8: Confusion matrix for the accuracy results with $\ell_{\text{min}} = 1$ and $\ell_{\text{max}} = 4$ of the "Digits" data set from the scikit-learn API.

The complexity for all four data sets appears to behave logarithmically. This is because the dimension only has a logarithmic influence on the complexity of sparse grids, while the level still influences it exponentially (see equation 2.33 and 2.40).
An interesting data set for further analyzation is the "Digits" data set, whose classes each represent an arabic digit and whose values describe an pixelated image of its corresponding class or digit. Since humans can intuitively map images of digits to its actual number, it is interesting to see how a computer algorithm tackles this problem. To exactly see, which digits were recognized correctly and which digits were confused with each other, a *confusion matrix* can be build (see Figure 4.8). Interesting to note are here, that digits 3, 8 and 2, 7 were confused with each other quite regularly but not necessarily symmetrically, i.e. 2 was never mistaken for 7, while 7 was mistaken for 2 31% of the time. Also the algorithm seems to have some trouble generally mapping the digits 1, 3, 5, 8 and 9 correctly.

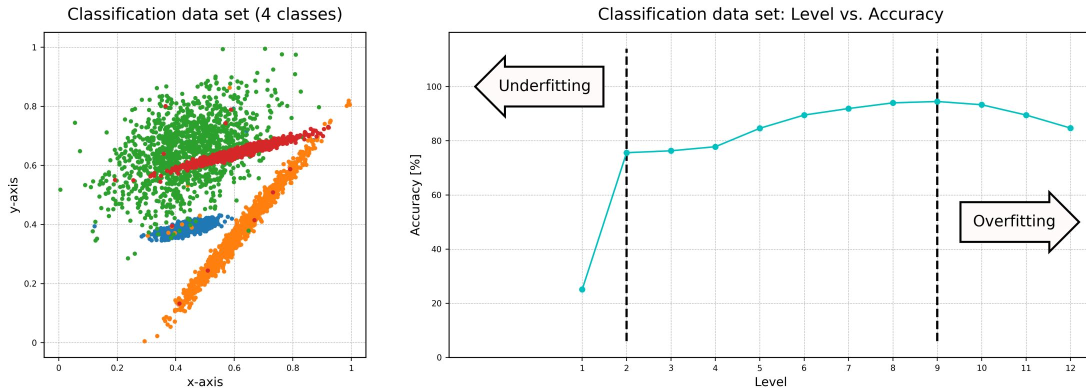


Figure 4.9: Example of under- and overfitting for the classification task with $\ell_{\text{min}} = 1$ based on a "Classification" data set with 5000 samples and 80% learning percentage. Underfitting only really occurs for the case of $\ell_{\text{max}} = 1$, where the algorithms does not have any relevant information about the classes and simply maps every sample onto the same class. Overfitting occurs for $\ell_{\text{max}} = 9$ and higher, because the high amount of hat functions makes the algorithm focus too much on unimportant information.

4.1.4 Under- and Overfitting

As briefly explained in chapter 2, section 2.3, a regular problem for machine learning tasks is *under-* and *overfitting*. For all previous examples in section 4.1.1 and section 4.1.3, underfitting occurred whenever the maximal value or *ceiling* was not already hit for a low enough level. Overfitting could not be seen for any example yet. However if a sparse grid with a high enough level would be used for a classification task, the high amount of hat functions would result in an overestimation of certain noise samples. This then would be the reason for overfitting (see Figure 4.9). To remedy this problem, the λ-parameter of the underlying density estimation could be adjusted accordingly to return a certain "smoothness" to the density function despite its many hat functions.

4.1.5 Learning to testing ratio

For all previous examples in this section, the *learning percentage* of an exemplary data set was always set to 80%. To get an understanding of how this percentage impacts accuracy of a classification task in general, some examples with different learning percentages are given in Figure 4.10. Note that there seem to be no significant changes in accuracy when changing the ratio of learning to testing data. Interesting however are some fluctuations for which learning percentage the accuracy is the highest, e.g.

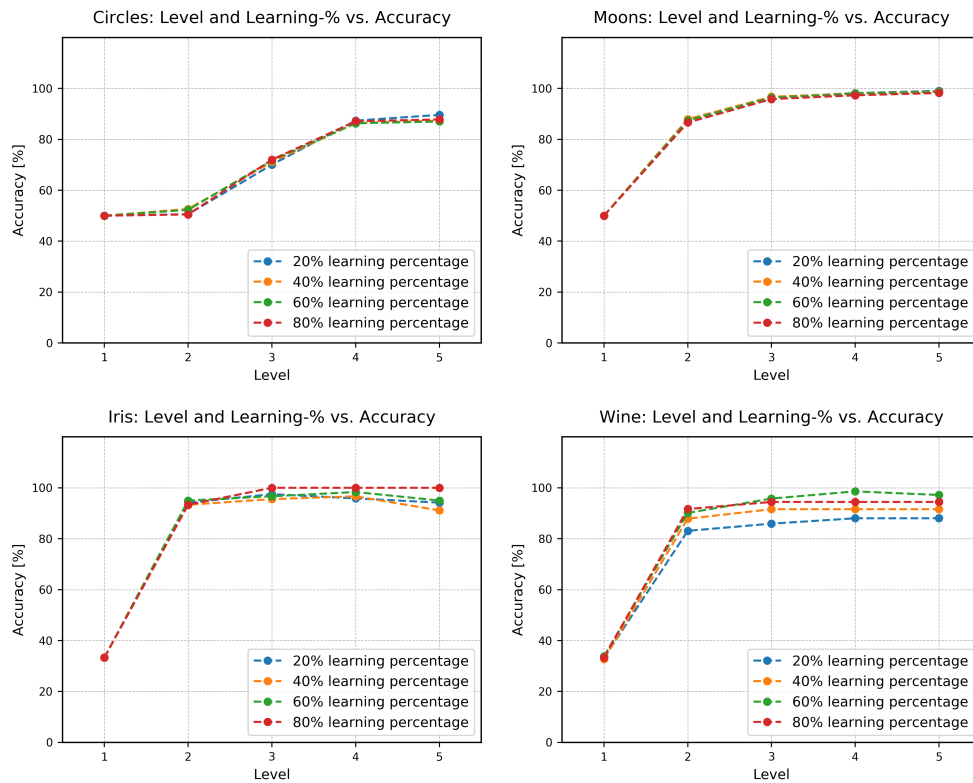


Figure 4.10: Comparison of the accuracies for the classification tasks "Circles" and "Moons" with 5000 samples each, "Iris" and "Wine" with varying learning percentages.

for the "Wine" data set a 60% learning subset seems to result in a higher accuracy than 80%. One possible reason for this could be overfitting, which seems to happens for 80% already at level 4.

4.2 Clustering

Since for a clustering task there is no information about its input data set D available besides the sample points themselves, results are hard to verify and analyze. One possibility to calculate the accuracy of the results from some clustering task is to remove the labeling of some known clustered data set, perform clustering on it and then compare the calculated clusters with the original labeling. This technique will be used for analyzation in this section. Again all five generated data set types mentioned at the start of chapter 4 will be used to show and explain the results of some clustering tasks by analyzing how certain input parameters like discretization levels ℓ and $\ell_{\max}$, the number of labels, the number of k nearest neighbors and the cutting threshold t_c affect output accuracy, measured in the percentage of correctly clustered sample points, and complexity, measured in the time for the algorithm to run.

4.2.1 Full vs. Sparse grids

As already explained for classification in section 4.1.1, full grids approximate sample points near the corners better than sparse grids do. Because of that it can be interesting to see how a clustering algorithm labels clusters near the corners of the domain $\bar{\Omega}^d$ and if full grids perform relatively better that sparse grid in comparison to classification tasks. The two following figures (see Figure 4.11 and Figure 4.12) show the accuracy and complexity for clustering tasks of some data sets with 5 nearest neighbors and a cutting threshold of $t_c = 0.25$ each for varying discretization levels. The minimal level is 1 in every example. The first two data sets contain 2500 and the last two 1000 samples.

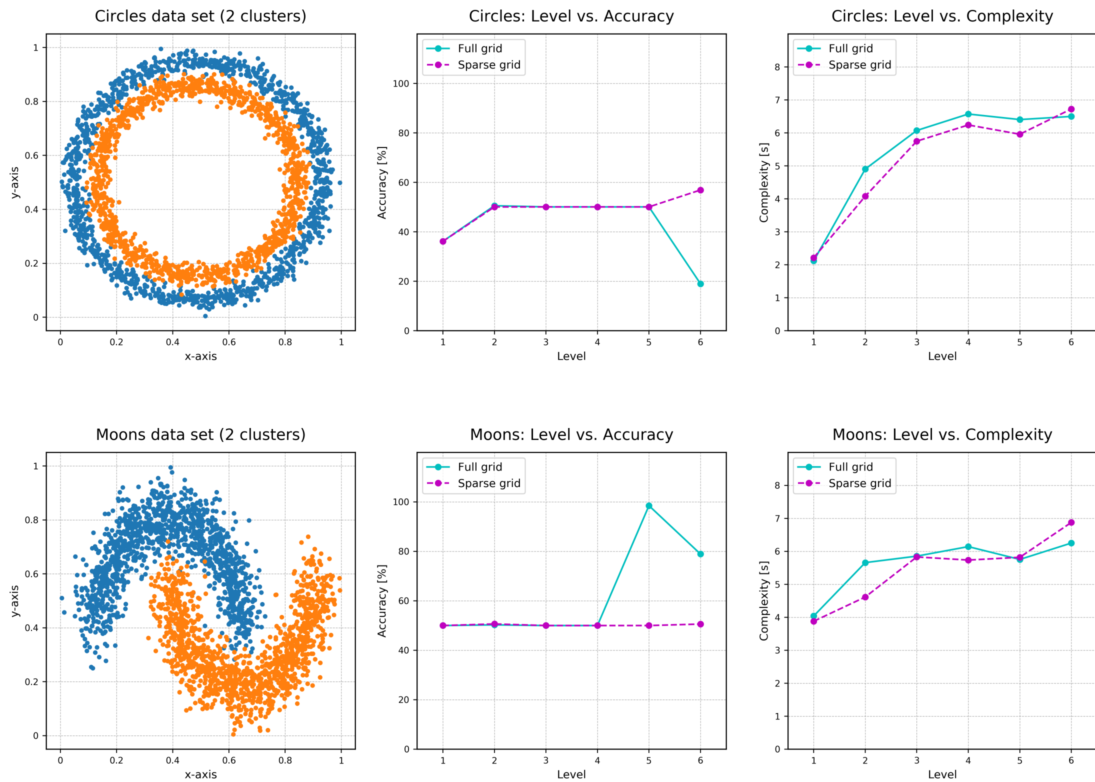


Figure 4.11: Accuracy and complexity based on discretization levels ℓ and ℓ_{max} between 1 and 6 of the clustering tasks for data sets "Cirlces" with 5% noise (top) and "Moons" with 15% noise (bottom).

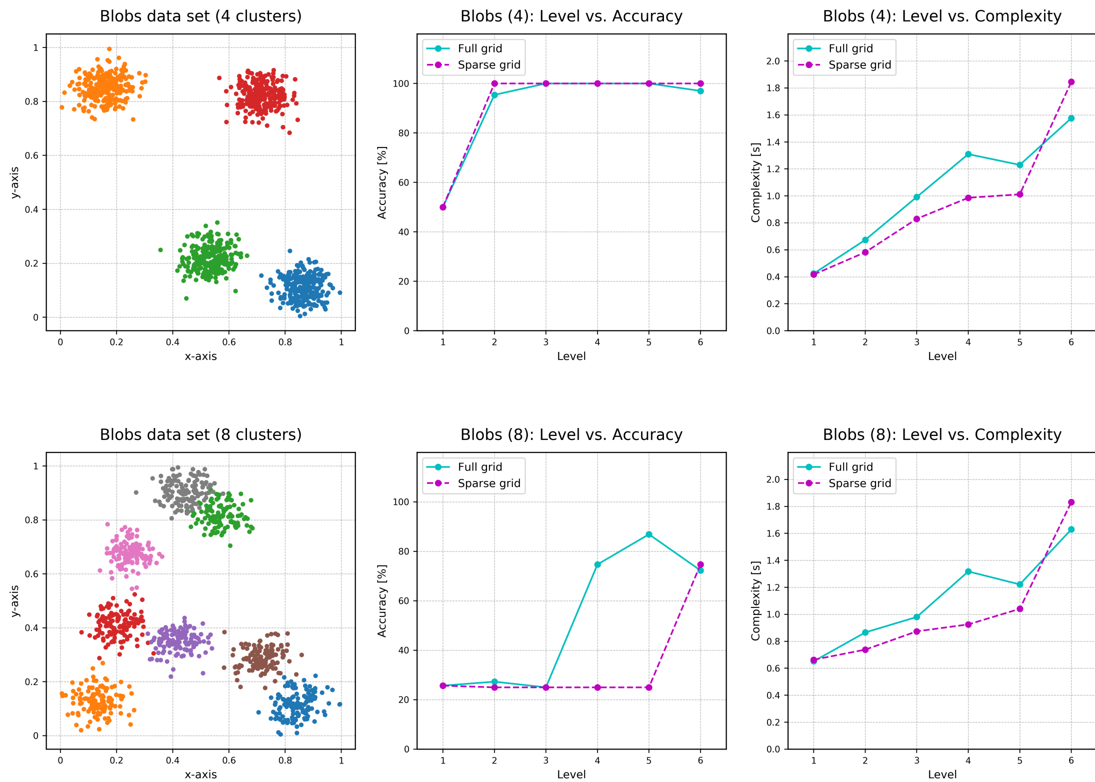


Figure 4.12: Accuracy and complexity based on discretization levels ℓ and ℓ_{max} between 1 and 6 of the clustering tasks for data set "Blobs", once with 4 classes (top) once with 8 classes (bottom).

General observations

If two clusters are too close to each other, the algorithm is not able to detect where it should cut certain edges. For Figure 4.11 this can clearly be seen for the "Circles" data set; the two concentric circles are interpreted as one and the algorithm is never able to achieve a higher accuracy than approximately 50%. At some level overfitting occurs for the full grid task and the algorithm overestimates the number of clusters to calculate. However if the level is high enough and the algorithm is able to detect the correct edges to cut, accuracy can near 100% as seen for the "Moons" example at full grid level 5 in the same figure. Full grids generally seem detect the right edges at an lower level than sparse grids, but are also prone to overfitting much earlier.
When looking at the two "Blobs" examples in Figure 4.12, the algorithm can clearly detect clusters with a high enough distance between them as seen in the example with only 4 labels. However when clusters are interleaving, the algorithm has difficulties to decide, which edges to cut and most of the time creates new bigger clusters, which then contain multiple true clusters. Figure 4.13 and Figure 4.14 show the differences of the clustering tasks for a full and sparse grid, both with a discretization level of 6, based on the "Blob" data set with 8 clusters. At level 5 overfitting already occurs for the full grid as seen in the graph of Figure 4.12, bottom mid, and by the many different computed clusters in Figure 4.14, bottom left. For the same discretization level the clusters computed with a sparse grid begin to be mostly correct, with the originally four interleaving clusters building only two computed ones. Also some clusters near the corners seem to have relatively many detected noise samples in comparison to other clusters for the sparse grid, which originates from their worse approximation of corner points, but does here not really impact the resulting accuracy, because the chosen discretization level is high enough.
Complexity for all two-dimensional examples still seems to hold to $\mathcal{O}\left(2^{2\ell}\right)$ for full and $\mathcal{O}\left(2^{\ell_{\max}} \cdot \ell_{\max}\right)$ for sparse grids (see equation 4.1 and 4.2). However one thing to note is that in comparison to classification, the number of samples heavily impacts the complexity of an algorithm, because every clustering task has to review n^k edges, whereas n is the number of samples and k the number of outgoing edges of every sample or rather the number of nearest neighbors.
An also interesting observation is that at level 6 and for dimension 2, the sparse grid complexity seems to exceed the full grid complexity for every example. To find the reason for this further research has to be carried out.

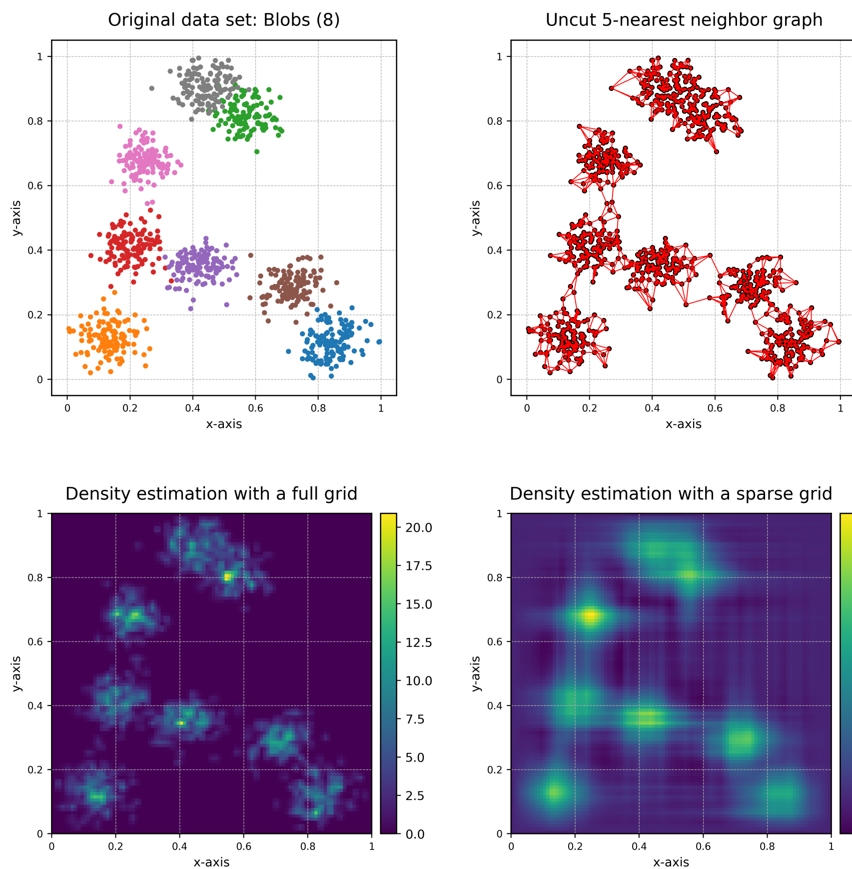


Figure 4.13: The "Blobs" example with 8 clusters from Figure 4.12, bottom. Located at the top right is the corresponding 5-nearest-neighbors-graph, whose edges haven't been cut yet. The corresponding density estimation with $\ell = 6$ based on a full grid and with $\ell_{\min} = 1$ and $\ell_{\max} = 6$ based on a sparse grid can be seen at the bottom left and right.

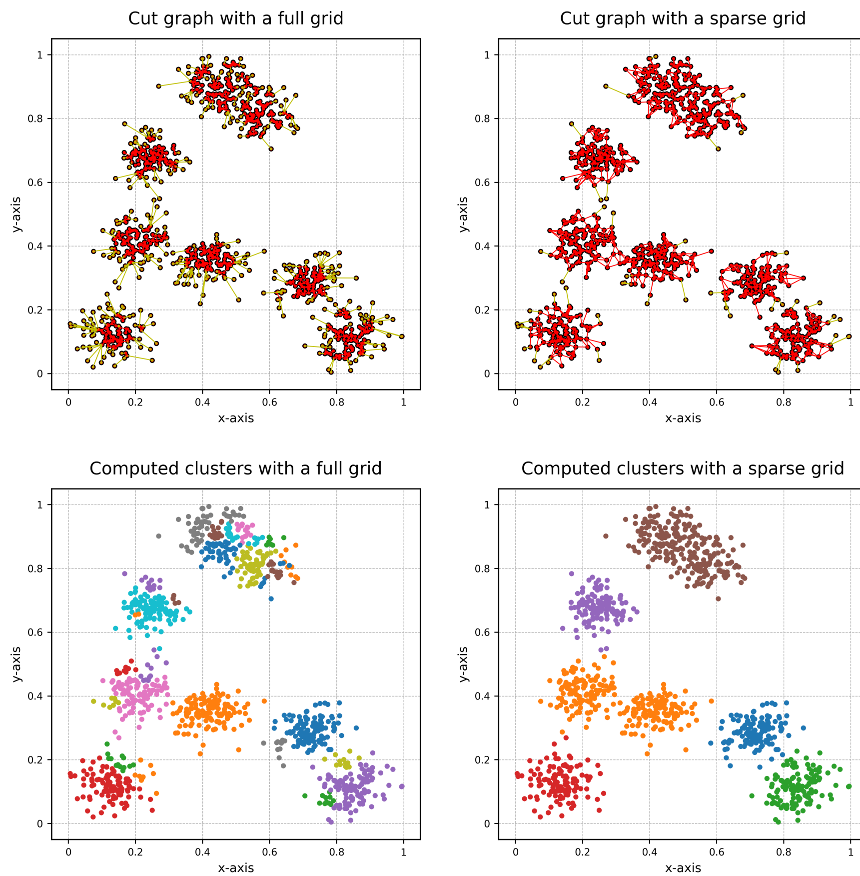


Figure 4.14: Continuation of the example stated in Figure 4.13. With the previously obtained density estimations, the uncut 5-nearest-neighbor-graph from before was cut for each one of them (top). With this the connected components or clusters are computed (bottom).

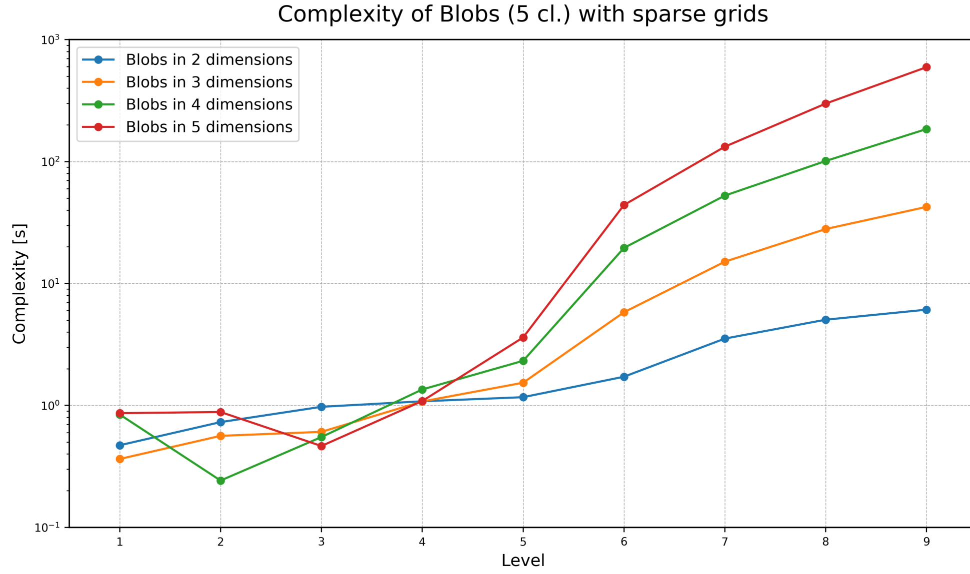


Figure 4.15: Comparison of the the complexity of four different "Blobs" data sets, each with 1000 samples, 5 clusters, 5 nearest neighbors, $t_c = 0.25$, $\ell_{\text{min}} = 1$ and a different dimension d, whereas $2 \leq d \leq 5$.

4.2.2 Clustering in more dimensions

With the equation for the number of grid points of sparse grids (see equation 2.33 and equation 2.40), the complexity of a clustering task is assumed to rise exponentially with increasing discretization level ℓ_{max} and logarithmically with rising dimension d. Figure 4.15 shows this based on the example of some "Blobs" data sets in dimensions 2, 3, 4 and 5 with 5 classes each. Interesting is here however that aforementioned relation of ℓ_{max}, d and the complexity only seems to hold for $\ell_{\text{max}} \geq 4$, seeing as the complexity for $\ell_{\text{max}} < 3$ seems to be generally lower for higher dimensions, with the exception of slight fluctuations for dimension 4 and 5 at level 1 and 2. One possible reason for the inconsistency before and after level 4 could be related to the relatively high spacing between clusters in high dimensions for only five possible labels: With such a small amount of labels it is more likely at lower levels that some edges were omitted than it is on higher levels. This consequently means that up until level 4, the evaluation of edges is more expansive than the evaluation of the underlying sparse grid.

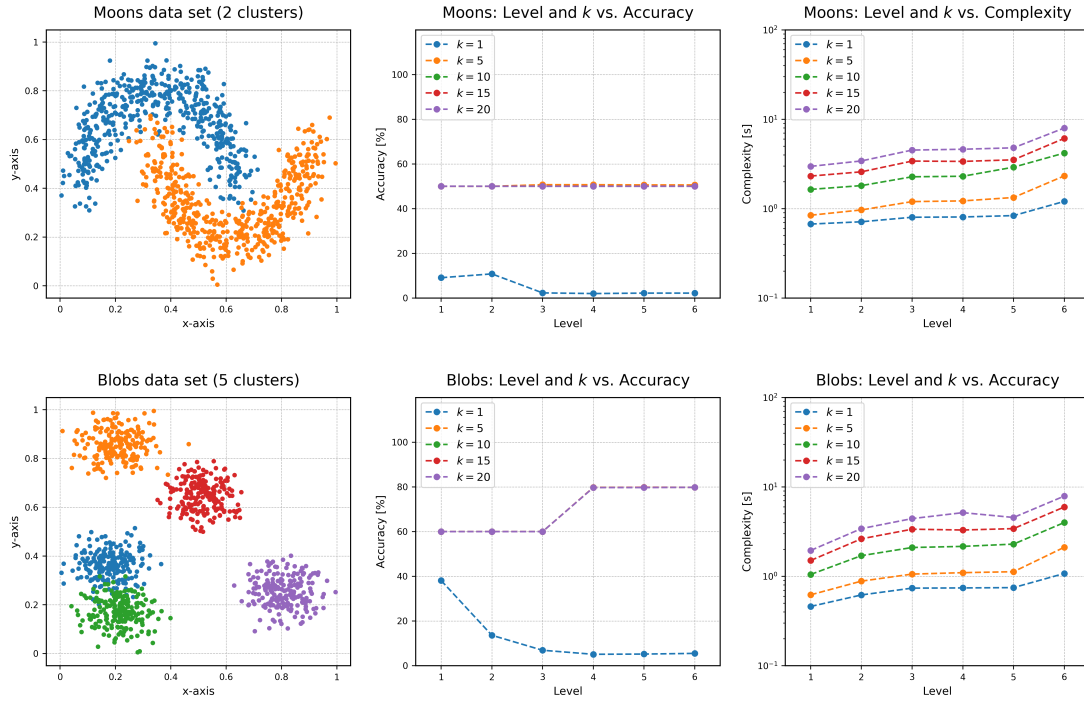


Figure 4.16: Accuracy and complexity based on discretization level $\ell_{\max}$ between 1 and 6 and the number of k nearest neighbors for the clustering tasks of generated data sets "Moons" with 15% noise (top) and "Blobs" with 4 clusters (bottom).

4.2.3 Varying number of nearest neighbors

As explained in section 4.2.1 does the number of k nearest neighbors affect the complexity of a clustering algorithm additionally by n^k, whereas n is the number of samples in the input set D. If and how k can also have an impact on the accuracy will be discussed in this section. Figure 4.16 displays accuracy and complexity based on discretization level and k neighbors for the clustering tasks of two exemplary data sets "Moons" and "Blobs" with 4 clusters, each with 1000 samples. Every task has parameters $\ell_{\min} = 1$ and $t_c = 0.25$. For both data sets, the linear additional complexity of n^k seems to hold true, since there is no exponential rise in complexity for higher k-values. The number of nearest neighbors however seems to have little effect on the accuracy for both examples, with the exception that the accuracy for $k = 1$ seems to near 0% fairly quickly. A possible reason for this could be that every edge essentially connects one part of a cluster with the corresponding other one, resulting in overfitting as early as $\ell_{\max} = 1$.

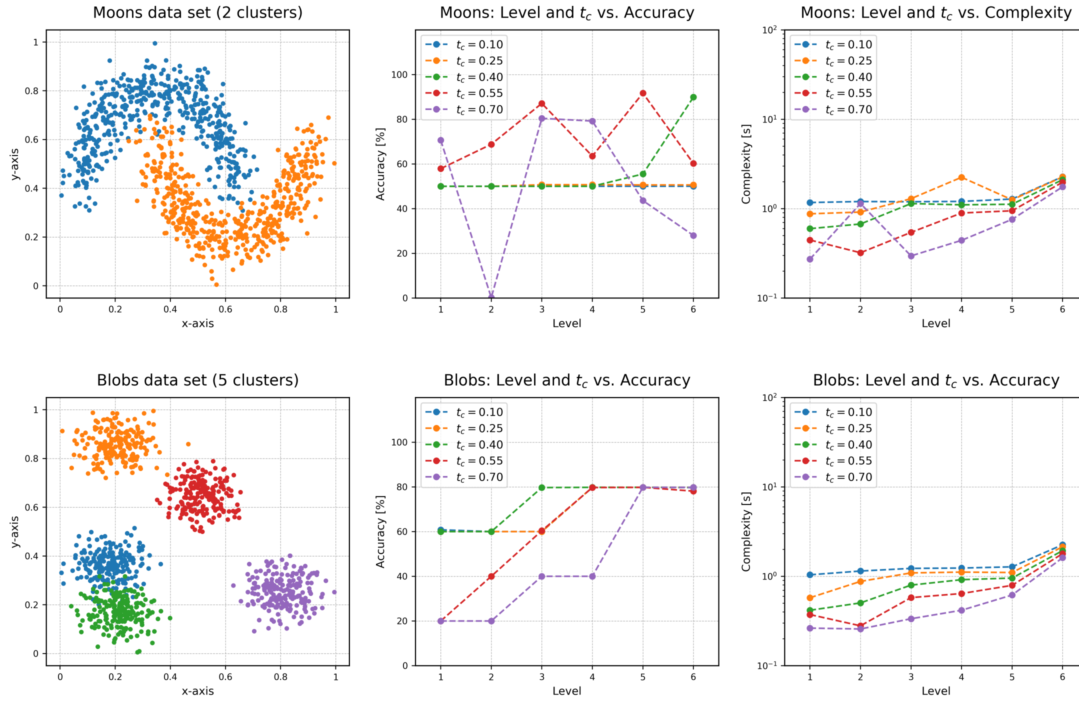


Figure 4.17: Accuracy and complexity based on discretization level $\ell_{\max}$ between 1 and 6 and the cutting threshold t_c for the clustering tasks of generated data sets "Moons" with 15% noise (top) and "Blobs" with 4 clusters (bottom).

4.2.4 Varying cutting threshold

Figure 4.17 again displays accuracy and complexity of the same two examplary data sets from Figure 4.16, with the exception of varying cutting threshold t_c and static number of neighbors $k = 5$ for every clustering task. When looking at the accuracy of both examples, it appears that extremely high or low thresholds result in an overall worse performance than thresholds at around 40%, whereas higher values of t_c let overfitting occur at lower levels. Complexity seems to decrease with rising t_c-value; a possible explanation for this being that a higher value results in more edges to be cut and therefore less edges for depth-first-search to traverse to detect the clusters.
An interesting observation is the heavy fluctuation of both accuracy and complexity for the "Moons" example at level 2 for $t_c = 0.7$. Here accuracy suddenly drops to nearly 0%, before rising again sharply at the next level. What happened here is that the algorithm probably detected one cluster for every sample, hence resulting in as many separate depth-first-search traversals, which explains the sudden rise in complexity.

5 Conclusion

This chapter provides a summary of all important achieved results in chapter 4, as well as presenting possible improvements for the implementation discussed in chapter 3 and giving insight into related works.

5.1 Summary

The implementation of the supervised and unsupervised machine learning tasks classification and clustering into the *sparseSpACE* framework shows, that with the use of sparse grids, higher dimensional machine learning tasks can be computed in a reasonable amount of time, while also approximating a corresponding full grid solution fairly well. This is exceptionally valuable for real-life machine learning task, since those are usually very high dimensional, e.g. learning the structure of an image, where every pixel represents one dimension. Also since unsupervised learning is usually more complex than supervised learning, the reduction in complexity that comes with the use of sparse grids makes them even more appealing for those unsupervised learning tasks.
As the results in chapter 4 show, the complexity of full grids for dimensions $d \leq 3$ and a low enough discretization level is generally still feasible, so that the use of sparse grids is preferable for dimensions $d \geq 4$.

5.2 Outlook

The implementation discussed in chapter 3 works as intended, but is still limited by certain factors, which provides some room for improvement. Mainly only data sets, whose samples all consist of the data type float, can be used as input for the implementation. So a possible addition could be to either map arbitrary data types of input sets to the float data type or to implement the support of more data types for the machine learning algorithms.
Some other limiting factors are derived directly from the harder to define unsupervised learning tasks. The presented implementation uses a rather intuitive approach to decide, which edges of a full nearest-neighbor-graph should be cut to receive the resulting

clusters. A possible improvement sees the adaptive selection of an *edge-cutting-threshold* according to the surrounding density or a more precise evaluation of the density of an edge, e.g. computing the average density of a valley between two points. In the same scope, the best relation between the cutting threshold and the number of nearest neighbors in terms of accuracy and complexity could be determined automatically for specific scenarios to maximize the performance. Also since the implementation of the clustering algorithm seems to have a relatively high complexity for cutting edges compared to evaluating the sparse grid for tasks with discretization levels smaller than four, a general code optimization in terms of run time is worth considering.
Other possible improvements include changes and additions to the underlying density-estimation-implementation, like the use of spatially adaptive sparse grids, which was implemented and also integrated into the `DEMachineLearning` wrapper in 2020 by Markus Fabry [14].

List of Figures

2.1 The standard hat function $\Phi(x)$ plotted in the domain $[-2,\ 2]$. 2

2.2 Example of nodal basis function space V_3 with level 3 and hat functions $\Phi_{3,i}(x)$ on grid points $x_{3,i}$ with $i \in I_3 := [1,\ 7]$. The boundary points are indicated with thin dotted black lines on the left and right borders. . . . 4

2.3 Example of a level 3 grid interpolated function $u_3(x)$ with weights $w_{3,i}$ (indicated by vertical gray dotted lines) and the nodal basis function space V_3. The boundary points are indicated with thin dotted black lines on the left and right borders. 4

2.4 Example of a 2-dimensional grid with level 4 in the first and level 2 in the second dimension. The grid points $x_{(4,2),\vec{i}}$ have the index set $I_{(4,2)} := ((1,\ 1),\ \ldots,\ (15,3))$. 6

2.5 Example of the tensor product approach to generate the 3D base functions $\Phi_{(2,1),(1,1)}(x)$, $\Phi_{(2,1),(2,1)}(x)$ and $\Phi_{(2,1),(3,1)}(x)$ (right) with the 2D base functions $\Phi_{2,1}(x)$, $\Phi_{2,2}(x)$ and $\Phi_{2,3}(x)$ (left) of level 2 and the 2D base function $\Phi_{1,1}(x)$ (middle) of level 1. 6

2.6 Example of hierarchical sub function space W_3 with level 3 and hat functions $\Phi_{3,i}(x)$ on grid points $x_{3,i}$ with $i \in I_{\text{odd},3} := [1,3,5,7]$. 8

2.7 Side by side comparison of the construction of the function space V_ℓ with the nodal approach (right) and the hierarchical approach (left) [taken from 5]. 8

2.8 Example of a level 3 grid interpolated function $u_3(x)$ (indicated by the bold dotted black line) with weights $w_{3,i}$ (indicated by vertical gray dotted lines) and the hierarchical basis function space V_3 (right). The sub function spaces W_k with $k \in [1,\ 3]$ summed up together build the level 3 function space V_3 (left). 9

2.9 Example of the construction of all 2D sub function spaces $W_{\vec{k}}$ with $\vec{k} \in [(1,\ 1),\ (3,\ 3)]$ for the 2D function space $V_{(3,3)}$ with level 3 using the tensor product approach. 11

2.10 Example of the 9 sub spaces for levels $|\vec{\ell}|_\infty \leq 3$ (left, gray and black grids) which together form the full grid function space V_3 and the corresponding sparse grid space $V_3^{(s)}$ (right), which consists of all 5 sub spaces with levels $|\vec{\ell}|_1 \leq 4$ (left, black grids above the dashed line) [taken from 6]. 13
2.11 Example of a sparse grid $\Omega_{1,3}^{(c)}$ with $\ell_{\min} = 1$ and $\ell_{\max} = 3$ (left) constructed by the linear combination of five component grids. Those are added whenever $q = 0$ and therefore $\alpha_q = 4$ and subtracted whenever $q = 1$ and therefore $\alpha_q = 3$ (right) [taken from 4]. 14
2.12 Example of all two-dimensional nodal based regular grids from level 1 to level 4 with boundaries. To create a sparse grid $\Omega_{1,4}^{(c)}$ with this function space, all blue component grids are added for the hyperplane 0 and all red component grids are subtracted for the hyperplane 1 [taken from 6]. 15
2.13 Example of three adaptive refinement steps of a sparse grid without boundaries. On a level 2 sparse grid, the most left point is selected for further refinement (left) and all its surrounding children added (middle). After yet another refinement step the missing path points (marked in gray) from previous discretization steps must also be added (right) [taken from 6]. 16
2.14 Example of a learning algorithm charged with the task of creating a curvature from some given points. On the left it cannot grasp the shape of the function (underfitting). In the middle it adapts the function well to the given points. On the right it assigns the single samples too much weight and estimates the curve incorrectly (overfitting) [taken from 7]. . 17
2.15 Example of the construction of a density function with a sparse grid of level 3. The data set "Circles" (top left) [taken from 10] serves as input S for the density estimation function. Sample points are mapped onto certain sparse grid points (top right). The density function $\hat{f}_{\text{sgrid},17}$ (bottom) is constructed with the constraint that equation 2.49 must hold for the basis functions on every grid point of the sparse grid. 20
2.16 Example of a given "Moons" training set set D_{train} (left) [taken from 10], which is split into its sub training sets $D_{\text{train},1}$ and $D_{\text{train},2}$ (right). 22
2.17 Example of performing density estimation on sub training sets $D_{\text{train},1}$ and $D_{\text{train},2}$ of the initial training set D_{train} (see Figure 2.16) independently to construct the mapping function $\hat{g}(\vec{x})$. 22

2.18 Example of the density distribution of a one-dimensional input set D. All input points x_i with $\hat{f}(x_i) \geq t_d$ (marked in yellow) are filtered into $R(t_d)$, which consists of two connected components. Between those there is a valley of points $\tilde{x}_i \notin R(t_d)$ [taken from 12]. 24
2.19 Example of the procedure of clustering a "Moons" data set [taken from 10] (top left) with a density estimation based on a combination-technique sparse grid with $\ell_{\min} = 1$ and $\ell_{\max} = 6$ (top mid). With this the 15-nearest-neighbor graph is build (bottom left) and then cut into two connected components with a cutting threshold of $t_c = 0.25$ (bottom mid). Note that there was detected some noise (marked in green), which was added to its nearest corresponding cluster. After that each data point is assigned one of those two clusters with recursive depth-first-search (mid right). 25

3.1 Python pseudo-code of the basic implementation of the preprocessing step in the `Classification` class. 28
3.2 Python pseudo-code of the basic implementation of the learning step in the `Classification` class. 29
3.3 Python pseudo-code of the basic implementation of the labeling step in the `Classification` class. 29
3.4 Python pseudo-code of the basic implementation of the evaluation step in the `Classification` class. 30
3.5 Python pseudo-code of the basic implementation of the preprocessing step in the `Clustering` class. 31
3.6 Python pseudo-code of the basic implementation of the preprocessing step in the `Clustering` class. 31
3.7 Python pseudo-code of the basic implementation of the nearest-neighbor-graph-building step in the `Clustering` class. 32
3.8 Python pseudo-code of the basic implementation of the nearest-neighbor-graph-cutting step in the `Clustering` class. Divided into the two sub steps "finding clusters" (top) and "appending noise" (bottom). 33
3.9 Python pseudo-code of the basic implementation of the detection step for the connected components in the `Clustering` class. 34
3.10 Python pseudo-code of the basic implementation of the evaluation step in the `Clustering` class. 35

4.1 Accuracy and complexity based on discretization levels ℓ and $\ell_{\max}$ between 1 and 6 for the classification tasks of data sets "Cirlces" with 5% noise (top) and "Moons" with 15% noise (bottom). 38

4.2 Accuracy and complexity based on discretization levels ℓ and $\ell_{\max}$ between 1 and 6 for the classification tasks of data sets "Classification" (top) and "Blobs" (bottom) with 4 classes each. 39
4.3 Accuracy and complexity based on discretization levels ℓ and $\ell_{\max}$ between 1 and 6 for the classification tasks of data sets "Blobs" (top) and "Gaussian Quantiles" (bottom) with 10 classes each. 40
4.4 Accuracy and complexity based on discretization levels ℓ and $\ell_{\max}$ between 1 and 5 for the classification tasks of the three-dimensional data sets "Classification" (top) and "Blobs" (bottom) with 8 classes each. . . . 41
4.5 Example of a classification task with $\ell_{\min} = 1$ and $\ell_{\max} = 6$ for a two-dimensional "Gaussian Quantiles" data set with 10 classes and 80% learning data. Because of the sparse grid construction of all 10 density functions (top), circles of testing samples appear to be mapped with a rectangular shape (bottom right). 43
4.6 Example of the step-by-step construction of the density function for some "Circles" data set with the combination technique with $\ell_{\min} = 1$ and $\ell_{\max} = 4$. The result can be seen in the top right corner. 44
4.7 Accuracy (left) and complexity (right) of four from scikit-learn loaded static data sets. Accuracy is measured by the percentage of correct mappings of the testing data. Complexity is measured by the computation time of the classification task and displayed on a logarithmic scale. . . . 46
4.8 Confusion matrix for the accuracy results with $\ell_{\min} = 1$ and $\ell_{\max} = 4$ of the "Digits" data set from the scikit-learn API. 47
4.9 Example of under- and overfitting for the classification task with $\ell_{\min} = 1$ based on a "Classification" data set with 5000 samples and 80% learning percentage. Underfitting only really occurs for the case of $\ell_{\max} = 1$, where the algorithms does not have any relevant information about the classes and simply maps every sample onto the same class. Overfitting occurs for $\ell_{\max} = 9$ and higher, because the high amount of hat functions makes the algorithm focus too much on unimportant information. . . . 48
4.10 Comparison of the accuracies for the classification tasks "Circles" and "Moons" with 5000 samples each, "Iris" and "Wine" with varying learning percentages. 49
4.11 Accuracy and complexity based on discretization levels ℓ and $\ell_{\max}$ between 1 and 6 of the clustering tasks for data sets "Cirlces" with 5% noise (top) and "Moons" with 15% noise (bottom). 51
4.12 Accuracy and complexity based on discretization levels ℓ and $\ell_{\max}$ between 1 and 6 of the clustering tasks for data set "Blobs", once with 4 classes (top) once with 8 classes (bottom). 52

4.13 The "Blobs" example with 8 clusters from Figure 4.12, bottom. Located at the top right is the corresponding 5-nearest-neighbors-graph, whose edges haven't been cut yet. The corresponding density estimation with $\ell = 6$ based on a full grid and with $\ell_{\text{min}} = 1$ and $\ell_{\text{max}} = 6$ based on a sparse grid can be seen at the bottom left and right. 54

4.14 Continuation of the example stated in Figure 4.13. With the previously obtained density estimations, the uncut 5-nearest-neighbor-graph from before was cut for each one of them (top). With this the connected components or clusters are computed (bottom). 55

4.15 Comparison of the the complexity of four different "Blobs" data sets, each with 1000 samples, 5 clusters, 5 nearest neighbors, $t_c = 0.25$, $\ell_{\text{min}} = 1$ and a different dimension d, whereas $2 \leq d \leq 5$. 56

4.16 Accuracy and complexity based on discretization level ℓ_{max} between 1 and 6 and the number of k nearest neighbors for the clustering tasks of generated data sets "Moons" with 15% noise (top) and "Blobs" with 4 clusters (bottom). 57

4.17 Accuracy and complexity based on discretization level ℓ_{max} between 1 and 6 and the cutting threshold t_c for the clustering tasks of generated data sets "Moons" with 15% noise (top) and "Blobs" with 4 clusters (bottom). 58

List of Tables

4.1 Accuracy (top) and complexity (bottom) of the "Circles" data set (see Figure 4.1) for the two input parameters minimal level ℓ_{min} and maximal level ℓ_{max}. 45

Bibliography

[1] K. P. Murphy. *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012. ISBN: 0262018020.

[2] C. Zenger. “Sparse Grids”. In: *Parallel Algorithms for Partial Differential Equations* (1991), pp. 241–251.

[3] M. Griebel, M. Schneider, and C. Zenger. “A Combination Technique For The Solution Of Sparse Grid Problems”. In: *Iterative Methods in Linear Algebra* (1992), pp. 263–281.

[4] T. Gerstner and M. Griebel. “Sparse Grids”. In: *Encyclopedia of Quantitative Finance* (2008). URL: https://ins.uni-bonn.de/media/public/publication-media/sparsegrids_j8NLaMi.pdf?name=sparsegrids.pdf.

[5] M. Bader. *Algorithms for Scientific Computing – 1D Hierarchical Basis*. 2017. URL: https://www5.in.tum.de/lehre/vorlesungen/asc/ss17/hierbas_1D.pdf.

[6] D. Pflüger, B. Peherstorfer, and H.-J. Bungartz. “Spatially adaptive sparse grids for high-dimensional data-driven problems”. In: *J. Complexity* 26 (Oct. 2010), pp. 508–522. DOI: 10.1016/j.jco.2010.04.001.

[7] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. URL: http://www.deeplearningbook.org.

[8] B. Peherstorfer, D. Pflüger, and H.-J. Bungartz. “Density Estimation with Adaptive Sparse Grids for Large Data Sets”. In: *SDM*. 2014.

[9] D. Pfander, G. Daiß, and D. Pflüger. “Heterogeneous Distributed Big Data Clustering on Sparse Grids”. In: *Algorithms* 12 (Mar. 2019), p. 60. DOI: 10.3390/a12030060.

[10] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[11] C. Aggarwal and C. Reddy. *DATA CLUSTERING Algorithms and Applications*. Aug. 2013.

[12] A. Azzalini and G. Menardi. "Clustering Via Nonparametric Density Estimation: the R PackagepdfCluster". In: *Journal of statistical software* (Jan. 2013). DOI: 10.18637/jss.v057.i11.

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

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