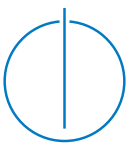
DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Integration of SGDE-based Classification into the SG++ Datamining Pipeline

Dominik Fuchsgruber



DEPARTMENT OF INFORMATICS

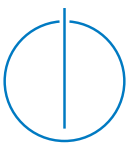
TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Integration of SGDE-based Classification into the SG++ Datamining Pipeline

Integration der SGDE-basierten Klassifikation in die SG++ Datamining Pipeline

Author:	Dominik Fuchsgruber
Supervisor:	Prof. Dr. rer. nat. habil. Hans-Joachim Bungartz
Advisor:	Kilian Röhner, M.Sc.
Submission Date:	16.08.2018



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

Munich, 16.08.2018 Dominik Fuchsgruber

Abstract

This thesis describes the integration of the sparse grid density estimation-based classification into the datamining pipeline of the SG++ toolbox. As the sparse grid density estimation can be reduced to solving a system of linear equations, a database to manage precomputed system matrix factorizations for different common scenarios was introduced. Furthermore, existing algorithmic modules were refactored, mainly by reorganizing the ownership of model state instances, such as the underlying grid objects and corresponding surplus vectors. Models for density estimation as well as classification were implemented as standalone tasks, therefore being made accessible to end-users. The new concepts are evaluated and compared to their previous implementation, showing that computational and memory complexity is maintained.

Zusammenfassung

Diese Arbeit befasst sich mit der Integration der auf der Dünngitter-Dichteschätzung basierenden Klassifikation in die Datamining Pipeline der SG++ Toolbox. Da die Dünngitter-Dichteschätzung auf das Lösen eines linearen Gleichungssystems zurückgeführt werden kann, wurde eine Datenbank eingeführt, welche vorab berechnete Zerlegungen von Systemmatrizen verschiedener häufig auftretender Szenarien verwaltet. Weiterhin wurden bereits existierende algorithmische Module angepasst, hauptsächlich durch die Reorganisierung der Verwaltung von zustandsbezogenen Instanzen, wie etwa den zugrundeliegenden Gittern und den zugehörigen Koeffizientenvektoren. Die Modelle für die Dichteschätzung und Klassifikation wurden so als eigenständige Funktionalität dem Endnutzer zugänglich gemacht. Zuletzt analyisert die Arbeit die neuen Konzepte und vergleicht sie mit der vorherigen Implementierung. Dabei wird insbesondere aufgezeigt, dass deren Komplexität sowohl hinsichtlich der Rechenzeit als auch des Speicherbedarfs erhalten geblieben ist.

Contents

1. Introduction

With the ever improving capabilities of technology and the availability of large datasets over the last decades, data-driven knowledge acquisition has risen in popularity and relevance. The extraction of information out of often huge data already plays a huge role in a broad variety of fields such as stock-market prediction, DNA sequencing and robotics, and is expected to have even more impact in the future. Since this knowledge discovery is usually not done manually, but should rather be automated, so-called Machine Learning techniques focus on algorithmic approaches to the subject.

One typical Machine Learning task is called classification, where the goal is to correctly classify new unseen data. The correlation between the data sample and its respective class label is inferred using a pre-classified training dataset. Despite appearing rather simple at first glance this problem statement can be applied to many real-world problems, one being for example cancer diagnosis.

Classification can be tackled employing different strategies, among the most popular being Random Forests, Nearest-Neighbor-based methods, Support Vector Machines and Deep Neural Nets. The approach described in this thesis tries to estimate the probability of a data sample belonging to a certain class by incorporating an approximation for each class-conditional probability density function and based on that predict the correct class label. In order to parametrize the density function, a grid-based approach is used: The density function is retrieved by combining different basis functions centered at the grid points.

However those methods typically do not scale well with the dimensionality of the data. Often computational effort and memory requirements even grow exponentially when dealing with high-dimensional datasets. Thus the data is sometimes preprocessed in order to reduce its dimensionality beforehand, which, however, may lead to a heavy information loss that could badly influence the classification performance. This issue is often referred to as the *curse of dimensionality* and also grid-based methods suffer from it severely.

Spatially Adaptive Sparse Grids provide a remedy to this problem to at least some extent, as they do not rely on a full grid structure to estimate the density function. Instead they try to adapt to the problem structure and only employ grid points in regions of interest, i.e. where the method is expected to benefit from those additional points. Since the computational expense of grid-based density estimation depends on

the number of grid points, which in the case of full grids grows exponentially with the grid dimension, a sparse grid structure can overcome the *curse of dimensionality* and make even high dimensional settings feasible.

The SG++ toolbox [11], created by Dirk Pflüger and developed at the chair of Scientific Computing at the Technical University of Munich and the Institute for Parallel and Distributed Systems at the University Stuttgart, impelements many sparse grid-based methods. Among those there has already been an approach to classification involving sparse grid density estimation as described by Peherstorfer [10]. This implementation however was rather conceptual and not yet integrated into the datamining pipeline of the toolbox, which is ought to provide easy access to the various data-driven functionality of the software. On the one hand many algorithmic components did not provide the necessary interface and had to be refactored, which will be discussed in Chapter 3. On the other hand the density estimation itself was not accessible to the user and instead directly incorporated into the classification modules.

This thesis describes the concepts and implementational changes necessary to overcome those issues and successfully integrate the sparse grid density estimation-based classification into the datamining pipeline. Furthermore, the performance of the modules is evaluated and compared to the previous implementation.

2. Theoretical Background

First of all, the theoretical background involving density estimation and classification methods as well as sparse grids themselves will be elaborated.

2.1. Machine Learning

As introduced in Chapter 1, Machine Learning approaches try to automatically retrieve information from data. Depending on the problem statement, a model is chosen that should be able to discover the underlying structure of a given dataset $\mathcal{S}$ and generalize well enough to make plausible predictions for new unseen data samples.

Usually however not the entire available data is used to train the model. Instead a portion of the dataset $\mathcal{S}$ is kept from the algorithm. Since during the training process the model can never learn from this dataset, it can be used to validate whether the model has generalized well by evaluating its performance on this dataset. Therefore this dataset is often referred to as *test data* $\mathcal{S}_{test}$. The performance of any Machine Learning approach should always be evaluated on the *test data*.

Even though the *test data* should never be involved in the training process, one might desire to validate the performance of the model during training nevertheless. Therefore, often the remaining portion of the dataset $\mathcal{S} \setminus \mathcal{S}_{test}$ is even further split into data that is actually used to train the algorithm $\mathcal{S}_{train}$ and data to validate the performance during training $\mathcal{S}_{val}$, which of course can not be involved in the training process as well. Note that the *test data* $\mathcal{S}_{test}$ and the *validation data* $\mathcal{S}_{val}$ are distinct from each other as the *test data* should only be used to evaluate the final performance of an already trained model.

In the following, if not explicitly mentioned otherwise, data samples will always be referred to as belonging to the *training data*. Consequentially M will describe the number of samples the *training data* consists of $M = |\mathcal{S}_{train}|$.

In Machine Learning, approaches are often categorized depending on the problem statement. Consider for example classification of handwritten digits. In addition to the feature vector, representing a single image, also the corresponding target, namely the digit that the sample represents, must be available to the algorithm. This fundamentally differs from a setting, in which the dataset $\mathcal{S}$ only consists of those feature vectors alone, for example when trying to find clusters in the dataset.

Settings, where those targets are part of the dataset, are called *supervised learning*. In contrast to that, we speak of *unsupervised learning* whenever these targets are not at hand. Commonly, even a third paradigm is mentioned: *Reinforcement learning* deals with problems, where in a given situation an agent should find the best action to take with respect to some rewarding function [1]. This thesis however will only deal with classification, one *supervised learning* setting that is based on another *unsupervised learning* technique, namely density estimation.

2.1.1. Density Estimation

We will first take a look at density estimation, an *unsupervised learning* task. Since no targets are involved, the dataset $\mathcal{S}$ can be formulated as consisting of only M d-dimensional feature vectors x_i, each representing one data sample.

$$\mathcal{S} = \{x_i\}_{i=1}^{M} \subset \mathbb{R}^d \tag{2.1}$$

For these samples, the basic assumption is, that they were drawn independently (and noisily) from a random variable X with unknown distribution. The goal of density estimation is to find a suitable function $\hat{f}(x)$, that approximates the true probability density function $f_X(x)$ of the distribution.

$$\hat{f}(x) \approx f_X(x) \tag{2.2}$$

One common method to formulate $\hat{f}(x)$ is, to sum kernel functions centered at the data samples x_i, which is described more detailed in [5]. However, whenever the resulting density function is to be evaluated, all kernel functions have to be considered. Therefore, the complexity of this approach scales linearly with the number of data samples, thus making settings involving especially large datasets intractable. In contrast to that, so-called sparse grids can be employed to estimate the density function, such that the complexity of its evaluation is independent of the data size. An in-depth elaboration of this method is provided in Section 2.3.

Regularization Generally speaking, in Machine Learning, the goal always is, to propose a function $\hat{f}(x)$ provides a good generalization in the context of the problem statement, rather than explaining the given data best. Usually, this is achieved by solving an optimization problem, where an algorithm tries to minimize a loss function, that only depends on already present training data.

This however does not necessarily lead to an approximation $\hat{f}$ that generalizes well to unseen data, but rather leads to a solution that only resembles the training data

very accurately. This issue is called overfitting and can be counteracted by employing regularization, where some degree of smoothness is enforced to the function $\hat{f}$.

Typically, this smoothness is achieved by implementing a regularization term into the loss function. It consists of a regularization operator Λ as well as a regularization strength λ. The latter describes the trade-off between explaining the training data $\mathcal{S}$ and smoothness of the function $\hat{f}$. It is one important so-called hyperparameter in most Machine Learning settings and has to be chosen carefully, when aiming for reasonable results.

2.1.2. Classification

A common *supervised learning* task is called classification, where each data sample x_i is associated with a categorical class label y_i. As the space of class labels is discrete, it can easily be mapped to a subset of the natural numbers $K = \{1, ..., k\} \subset \mathbb{N}$. In this *supervised learning* setting, the dataset can be formalized as a set of tuples of feature vectors and targets.

$$\mathcal{S} = \{(x_i, y_i)\}_{i=1}^{M} \subset \mathbb{R}^d \times K \tag{2.3}$$

The classification problem can then be formulated as finding an approximation $\hat{f}(x)$ for the underlying unknown function $f\colon \mathbb{R}^d \mapsto K$, that maps samples from the data space to their corresponding class label.

$$\hat{f}(x) \approx f(x) \tag{2.4}$$

There has been a great number of approaches to this problem, among the most popular being Random Forests, Support Vector Machines and Deep Neural Networks. The method this thesis focuses on however employs density estimation, as described in Section 2.1.1, to predict the class label of unseen data samples.

Density Estimation based Classification The main idea of this technique is to estimate the probability of observing a sample given a certain class label and then use Bayes' theorem to predict the class for a new data sample.

For each class $c \in K$, the so-called class-conditional probability density function $p(x \mid y = c)$ is estimated separately using only training samples $\mathcal{S}_c$ associated with this class c.

$$\mathcal{S}_c = \{x_i \in \mathcal{S} \mid y_i = c\} \tag{2.5}$$

The probability of a data sample belonging to a certain class c can be formulated by applying Bayes' theorem as mentioned before:

$$p\left(y = c \mid x\right) \propto p\left(x \mid y = c\right) p\left(y = c\right) \tag{2.6}$$

Note that the normalization constant $\left(p\left(x\right)\right)^{-1} = \left(\sum_{c' \in K} p\left(y = c', x\right)\right)^{-1}$, which is required in order for $p\left(y = c \mid x\right)$ to be a valid distribution, is the same for each class $c \in K$ and thus can be omitted, which can be seen in Equation (2.8).

One possible way to estimate the class prior $p\left(y = c\right)$ is to use the relative frequency of data samples associated with class c.

$$p\left(y = c\right) = \frac{|\mathcal{S}_c|}{|\mathcal{S}|} \tag{2.7}$$

The prediction function can then be formulated as finding the class label c that maximizes $p\left(y = c \mid x\right)$.

$$\hat{f}\left(x\right) = \underset{c \in K}{\arg\max}\, p\left(y = c \mid x\right) \tag{2.8}$$

2.2. Grid Based Interpolation

The technique described in Section 2.1.2 relies on the performance of the density estimation of the class-conditional probabilities. However, as hinted before in Section 2.1.1, common methods do not scale well with the dimensionality of the problem. Therefore, we introduce sparse grids as one alternative approach to density estimation that provides a remedy to the *curse of dimensionality* to some extent.

To get a grasp of the concept, consider the task of interpolating an unknown function $f\colon \Omega \mapsto \mathbb{R}$. In the following we restrict the function domain Ω to be the d-dimensional unit-hypercube, $\Omega := [0,1]^d$.

2.2.1. Full Grid Interpolation

To begin with, consider interpolating a function f on a full grid. In this case, usually the mesh width is chosen as 2^{-n}, thus resulting in a regular full grid of discretization level n with $N = 2^n - 1$ equidistant grid points in each dimension.

For the sake of simplicity, only the one-dimensional interpolation problem will be presented first and only later extended to a multi-dimensional domain. For convenience, the grid points will not be referred to by their Cartesian coordinates but rather indexed by $i \in \{1, ..., N\}$.

The interpolation task can be attempted by introducing basis functions with local support, each of which is centered at a different grid point. There are various choices for the type of basis function to employ, many of them being more or less suitable for

different scenarios. Usually in Machine Learning related problems however, the lack of prior knowledge enforces using piecewise d-linear basis functions, as they are the most easy ones to deal with. A common example is the standard hat function.

$$\varphi(x) = \max(1 - |x|, 0) \tag{2.9}$$

Nodal Basis In the nodal basis the basis function centered at grid point i only provides support between grid points $i-1$ and $i+1$. Thus, the i-th basis function can be obtained by dilatation and translation [11].

$$\varphi_i(x) = \left(2^{n-1}x - i\right) \tag{2.10}$$

The interpolant $p(x)$ is given as weighted sum of the basis functions φ_i, as can be seen in Figure 2.1. The weighting coefficients α_i are called surpluses [11].

$$f(x) \approx p(x) := \sum_{i=1}^{N} \alpha_i \varphi_i(x) \tag{2.11}$$

Hierarchical Basis Another option is to put the different basis functions in a hierarchical order. Basis functions at coarser levels provide broader support and thus are responsible for the overall structure of the interpolant. In contrast, basis functions at very fine levels only contribute to the function locally.

Instead of referring to grid points using only their index, now also their level is necessary to identify it. Therefore, we define a set of hierarchical indexes for a certain level l.

$$I_l = \left\{ i \in \mathbb{N} \colon 1 \leq i \leq 2^l - 1 \wedge i \mod 2 = 1 \right\} \tag{2.12}$$

A grid point can now be indexed by a tuple (l, i), that consists of its level $l \leq n$ and index at this level $i \in I_l$. Again, the normal hat function from Equation (2.9) is used as basis function, each of which will still be centered at another grid point (l, i) [11].

$$\varphi_{l,i} = \varphi\left(2^l x - i\right) \tag{2.13}$$

Furthermore, the hierarchical subspaces W_l can be defined as span of all basis functions $\varphi_{l,i}$ centered at grid points of level l.

$$W_l = \operatorname{span}\{\varphi_{l,i} \colon i \in I_l\} \tag{2.14}$$

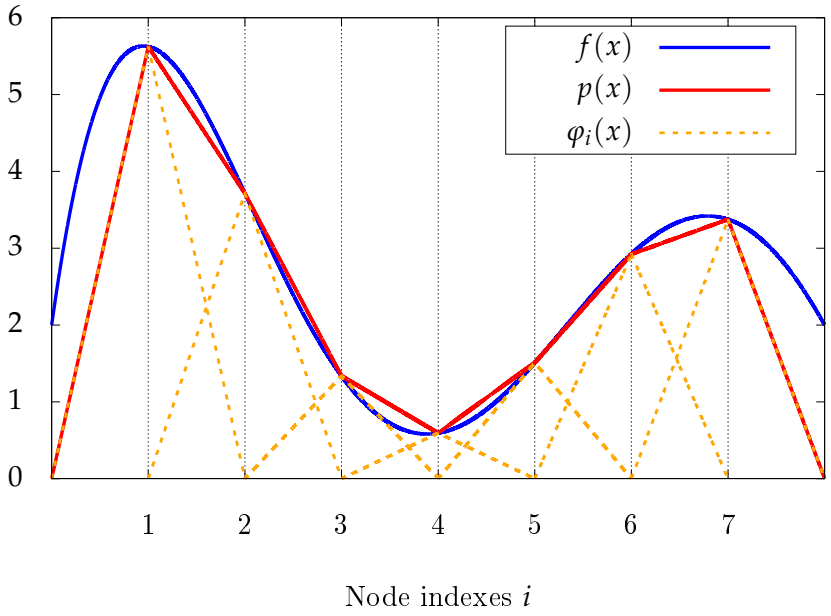


Figure 2.1.: Interpolant (red) of a polynomial function (blue) using a linear combination of nodal basis functions φ_i (orange) scaled by their surpluses α_i.

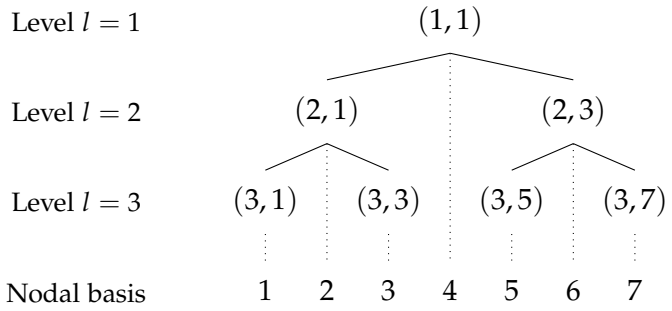


Figure 2.2.: Visualization of the hierarchical basis and their correspondence to the nodal basis (dotted lines). The nodes of the hierarchical basis are indexed by (l, i) where l is the level and i the index at level l respectively.

The entire space of piecewise linear functions on a full grid with discretization level n can be obtained by directly summing over all the hierarchical subspaces up to level $l \leq n$ [11].

$$V_n = \bigoplus_{l \leq n} W_l \tag{2.15}$$

Figure 2.2 gives a visual intuition of why the combination of hierarchical subspaces in fact yields the entire full grid. Each grid point in the hierarchical basis directly corresponds to an unique grid point in the nodal basis. When later dealing with sparse grids, the tree structure in which the hierarchical basis functions are organized, can be pruned or extended at certain nodes, which is called coarsening and refinement respectively.

Similar to interpolation using the nodal basis, the interpolant can be written as weighted sum of hierarchical basis functions. In contrast to the nodal basis however, the surplus vector as well as the basis functions now have to be indexed by their level and index at this level [11]. Figure 2.3 offers a visualization for interpolation on a full grid using the hierarchical basis.

$$f(x) \approx p(x) := \sum_{l \leq n, i \in I_l} \alpha_{l,i} \varphi_{l,i} \tag{2.16}$$

Higher Dimensions Previously only univariate settings were considered. Higher dimensional problems however can be derived easily, making use of a tensor product structure to describe the interpolant [11].

First of all however, notational aspects and common vector norms, that are necessary

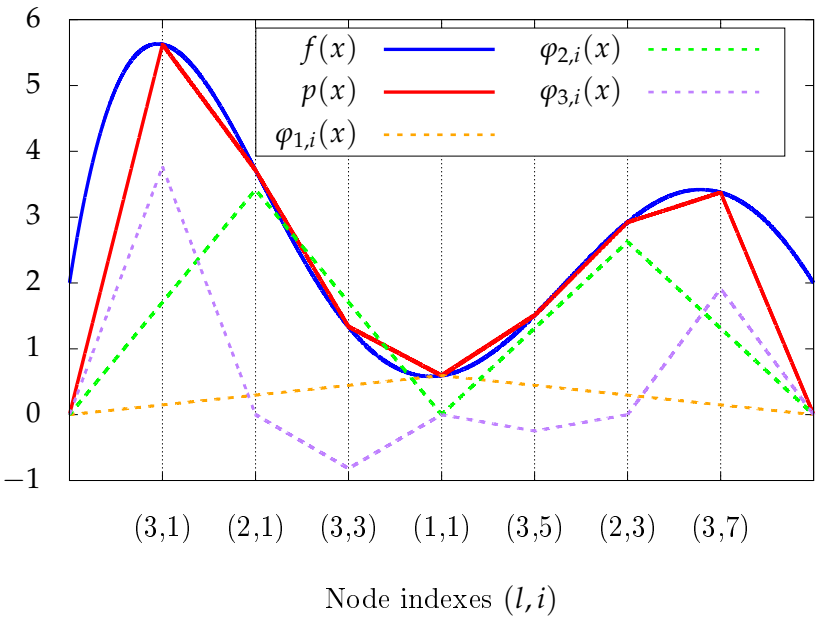


Figure 2.3.: Interpolant (red) of a polynomial function (blue) using a linear combination of hierarchical basis functions $\varphi_{l,i}$ (orange, green, purple) scaled by their surpluses $\alpha_{l,i}$.

to describe this construction, will be defined. Namely, those are the l_1-norm $|\vec{l}|_1$ and the maximum norm $|\vec{l}|_\infty$.

$$|\vec{l}|_1 := \sum_{j=1}^{d} |l_j| \quad \text{and} \quad |\vec{l}|_\infty = \max_{1 \leq j \leq d} |l_j| \tag{2.17}$$

Instead of only using a single level l and index i, now d-dimensional vectors $\vec{l}$ and $\vec{i}$ are required to identify a hierarchical basis function in the d-dimensional grid.

$$\varphi_{\vec{l},\vec{i}}(\vec{x}) := \prod_{j=1}^{d} \varphi_{l_j,i_j}(x_j) \tag{2.18}$$

The set of valid indexes for a given vector of levels can be defined in a straight forward fashion [11].

$$I_{\vec{l}} = \left\{ \vec{i} \colon 1 \leq i_j \leq 2^{l_j} - 1 \wedge i_j \mod 2 = 1 \wedge 1 \leq j \leq d \right\} \tag{2.19}$$

Multi-dimensional hierarchical subspaces can be derived analogously to Equation (2.14) [11]. Figure 2.4, taken from [11], offers a visualization of the two dimensional hierarchical subspaces for $|\vec{l}|_\infty \leq 3$.

$$W_{\vec{l}} = \operatorname{span} \left\{ \varphi_{\vec{l},\vec{i}}(\vec{x}) : \vec{i} \in I_{\vec{l}} \right\} \tag{2.20}$$

Similar to Equation (2.15), the full grid space with maximal discretization level n can be written as sum over those hierarchical subspaces [11].

$$V_n = \bigoplus_{|\vec{l}|_\infty \leq n} W_{\vec{l}} \tag{2.21}$$

Lastly, the interpolation itself can be reformulated like in Equation (2.16).

$$f(\vec{x}) \approx p(\vec{x}) = \sum_{|\vec{l}|_\infty \leq n, \vec{i} \in I_{\vec{l}}} \alpha_{\vec{l},\vec{i}} \varphi_{\vec{l},\vec{i}}(\vec{x}) \tag{2.22}$$

2.2.2. Sparse Grid Interpolation

When performing interpolation on a full grid, each basis function needs to be evaluated at least once. Therefore, the computational effort highly depends on the number of grid points, as each of them provides one basis function that offers support for the interpolant $p(\vec{x})$. In the case of regular full grids with discretization level n and

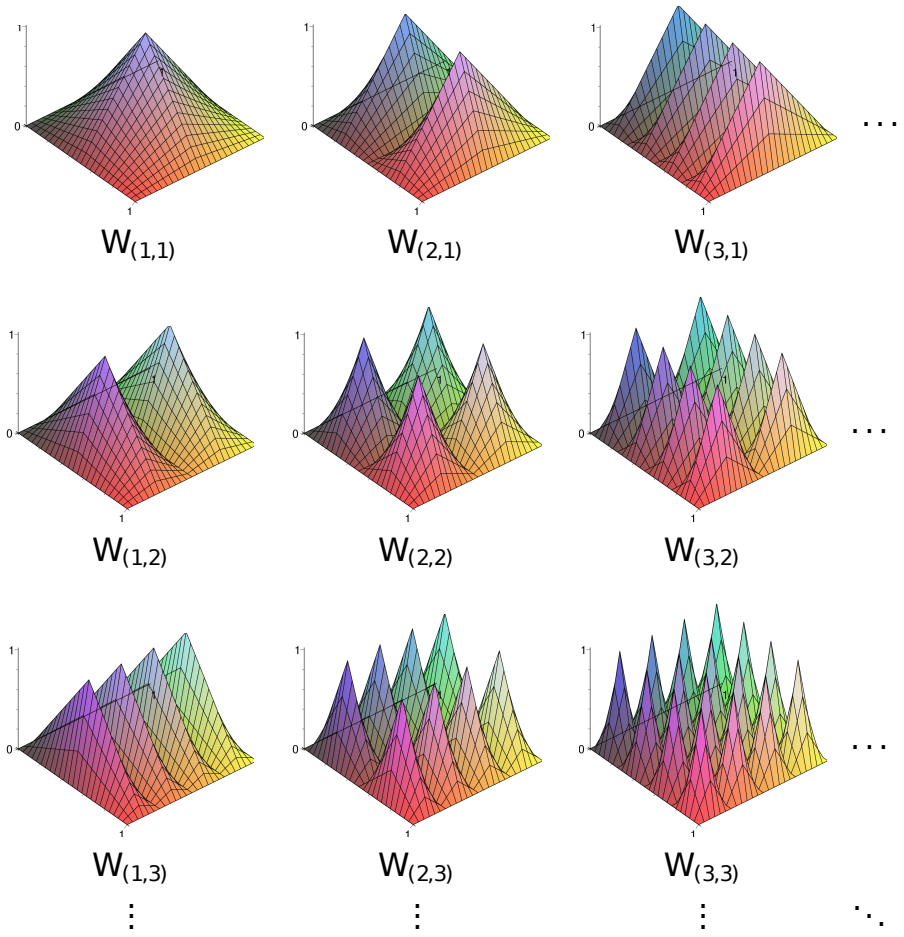


Figure 2.4.: Two dimensional hierarchical subspaces $W_{\vec{l}}$ with $|\vec{l}|_\infty \leq 3$. Figure taken from [11].

dimension d, the number of grid points grows exponentially with the dimensionality [11].

$$|\mathcal{G}_{full}| = (2^n - 1)^d \in \mathcal{O}(2^{nd}) \tag{2.23}$$

Thus, for high dimensional settings, interpolation on regular full grids suffers from the full *curse of dimensionality* and quickly becomes infeasible. Sparse grids introduce a way to drastically reduce the number of grid points, therefore making even tasks in higher dimensions tractable. The underlying basic concept again is to combine hierarchical subspaces $W_{\vec{l}}$, in order to retrieve the regular sparse grid space $V_n^{(1)}$. In contrast to full grids however, fine grained subspaces that only pay small contribution to the overall interpolant $p(\vec{x})$ are left out [11].

$$V_n^{(1)} = \bigoplus_{|\vec{l}|_1 \leq n+d-1} W_{\vec{l}} \tag{2.24}$$

Figure 2.5, taken from [11], depicts how the combination of coarse hierarchical subspaces results in a regular sparse grid of level $n = 3$.

The interpolant $p(\vec{x})$ for full grids, described in Equation (2.22), can easily be adapted to the sparse grid structure as well.

$$p(\vec{x}) = \sum_{|\vec{l}|_1 \leq n+d-1, i \in I_{\vec{l}}} \alpha_{\vec{l},\vec{i}} \varphi_{\vec{l},\vec{i}}(\vec{x}) \tag{2.25}$$

Note that the upper bound for the number of grid points contained in a sparse grid of level n, reduces to $\mathcal{O}(2^n n^{d-1})$, while the error increases ever so slightly for sufficiently smooth functions f [11] [4].

2.2.3. Spacial Adaptivity

Even though regular sparse grids significantly improve the tractability of high dimensional settings, the *curse of dimensionality* still prevails for a high number of dimensions and high sparse grid level. Also, even though sparse grids might not spend as many grid points in regions of low interest for the problem as full grids do, still their structure does not necessarily adapt well to the problem setting at all. There even might be regions where points are desired to populate the function domain more densely. To counteract these problems, yet another concept is introduced, namely spatial adaptivity. Using spatially adaptive sparse grids as described by Dirk Pflüger in [11], grid points can be added to or removed from the sparse grid, without needing to introduce a new level in each dimension.

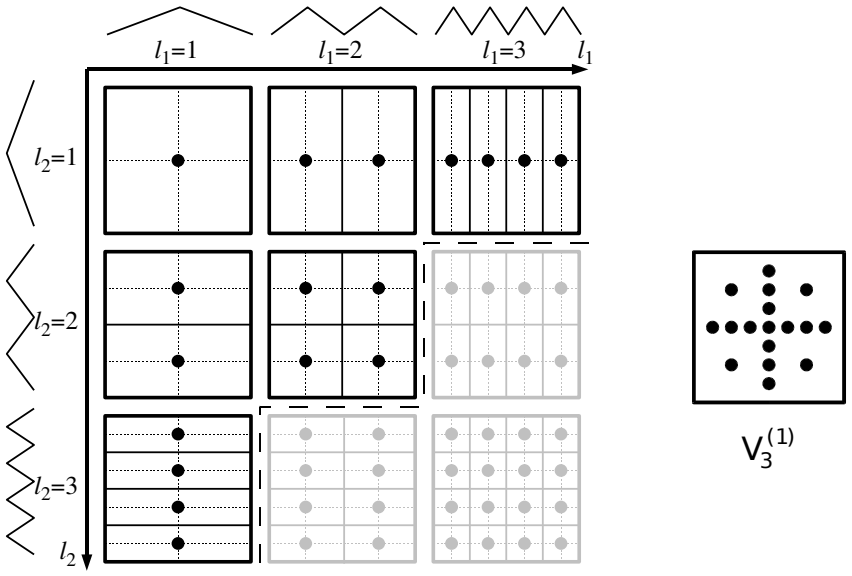


Figure 2.5.: Construction of a regular sparse grid of level $n = 3$ from different hierarchical subspaces. Adding the greyed-out subspaces as well would result in a regular full grid. Figure taken from [11].

This can be imagined as locally pruning or extending the hierarchical tree of grid points, that is depicted by Figure 2.2 for one dimensional settings. By employing a heuristic function, regions in which additional grid points might increase the model performance can be determined as well as regions, in which grid points are superfluous, as they do not contribute much to the overall solution.

Refinement The process of adding new points to the sparse grid is called refinement. After the application of a suitable heuristic, children are added to leaf nodes in the hierarchical tree of grid nodes. Often, performing refinement for only one grid point can lead to overfitting the target function, which in the case of interpolation might be desired, but when considering density estimation most likely negatively affects the accuracy. Thus, refining multiple grid points during each refinement step simultaneously, usually provides better results in these settings. For a more detailed examination of refinement strategies and heuristics for refinement, refer to [7].

Another noteworthy aspect is, that many sparse grid based algorithms require the hierarchical tree-like structure of grid points to be traversable in both directions. Therefore when adding grid points as children to an existing leaf node in one dimension, parent grid points might have to be created in order to preserve a valid hierarchical

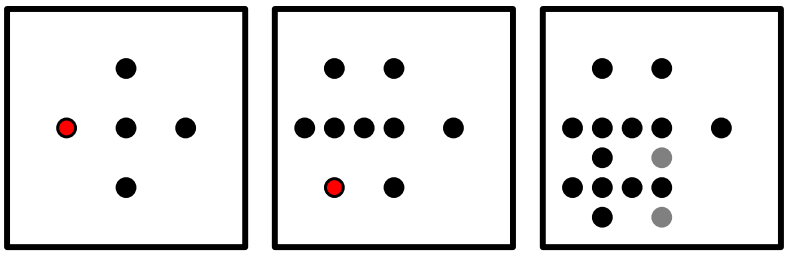


Figure 2.6.: Two refinements on a level two sparse grid. The refinement of the first point (red) does not require additional parent points in the hierarchical tree (left). The second refinement step however creates points that lack parents in the vertical dimension (middle). Those parent points (gray) are added to the grid as well, in order to preserve traversability for the hierarchical tree in both directions (right). Figure taken from [11].

structure in those dimensions as well. Figure 2.6 illustrates this process of recursively adding parents to newly introduced grid points. Usually in data-driven settings however, this kind of tree traversal is not a necessity.

Coarsening Since the computational effort highly depends on the number of basis functions, it also makes sense to remove grid points, from which the solution does not really benefit. This process is called coarsening and can be understood as pruning the hierarchical tree-like structure (see Figure 2.2) of the sparse grid. Again, the impact on the accuracy of the model depends on the heuristic that determines which and how many grid points should be coarsened during each step. An in-depth discussion can again be found in [7].

2.3. Sparse Grid Density Estimation

For simplicity, if not defined otherwise, a hierarchical basis will be implicitly assumed when talking about any sparse grid from here on. Furthermore, instead of referring to the basis functions and surpluses by a tuple $\left(\vec{l}, \vec{i}\right)$, they will be indexed using enumeration, similar to indexing nodal basis functions.

Next, an approach to density estimation as described in Chapter 2.1.1, that operates on sparse grids, is introduced. Given our training dataset $\mathcal{S}$, we can obtain a highly overfitted initial guess for our density function f_ϵ, by using Dirac delta functions centered at the data samples. In order to retrieve a more generalizing solution, regularization will be applied to this function, where Λ describes the regularization operator and λ the regularization strength respectively. The density function $\hat{f}$ can be obtained

by solving the optimization problem given by Equation (2.26) [10].

$$\hat{f} = \arg\min_{\tilde{f}} \int_{\Omega} \left(\tilde{f}(x) - f_{\epsilon}(x)\right)^2 \mathrm{d}x + \lambda \|\Lambda \tilde{f}\|_2^2 \tag{2.26}$$

As elaborated in [10], when applying this approach to the space of sparse grid basis functions, a system of linear equations can be derived as solution to the optimization task.

$$(R + \lambda C)\,\vec{\alpha} = \vec{b} \tag{2.27}$$

In this system, the matrix R represents the structure of the underlying sparse grid, as it is defined as $R_{i,j} = \langle \varphi_i, \varphi_j \rangle_{L_2}$, where $\langle \cdot, \cdot \rangle_{L_2}$ denotes the standard L_2 inner product. The regularization is represented by the matrix C, which is usually chosen to be the identity matrix, since it makes computational aspects much easier. As previously, $\vec{\alpha}$ describes the hierarchical surpluses of the sparse grid, i.e. the coefficients of the hierarchical basis functions φ. Lastly, the right-hand side b corresponds to the training data $\mathcal{S}$, since $b_i = \frac{1}{M} \sum_{j=1}^{M} \varphi_i(x_j)$ [10].

One thing to note is that the size of the left-hand side matrix $(R + \lambda C) \in \mathbb{R}^{n \times n}$ does not depend on the size of the training data, but rather the number of points contained by the underlying sparse grid. Since the computational complexity of solving this system is in $\mathcal{O}(n^3)$, efficient methods have to be considered in order to tackle sparse grid density estimation. One possible approach involves using conjugate gradients to iteratively find a solution. An advantage of this method is, that it does not require the entire matrix to be loaded into memory [10]. Another way to deal with the cubic complexity is to factorize the left-hand side system matrix beforehand and use the decomposition to efficiently solve the system afterwards. This will be referred to as Online-/Offline Splitting and is explained in Section 2.4.

2.4. Online-/Offline Splitting

As mentioned before, when dealing with sparse grid density estimation, a system of linear equations has to be solved. Since the left-hand side matrix $R + \lambda C$ is independent of the actual training data, factorizing the system matrix beforehand can be used to enhance the runtime performance of the density estimation drastically. The computation of the factorization usually has a runtime complexity of $\mathcal{O}(n^3)$ and will be referred to as offline step, since it can be performed without requiring any data whatsoever, i.e. the left-hand side only depends on the structure of the sparse grid itself as well as the regularization. When attempting to solve the system for any given right-hand side b, which can only be computed with data at hand, the computational complexity

decreases to $\mathcal{O}(n^2)$. This step will be called online step.

Adaptivity Since the left-hand side of the system depends on the grid structure, one aspect which has to be considered is spatial adaptivity as described in Section 2.2.3. In Machine Learning settings, typically any kind of prior knowledge is hardly ever available. Therefore, in order to achieve reasonable performance, it is inevitable for the sparse grid to be able to adapt to the given problem structure itself. This however can not be implemented into the offline step, as these structural changes depend on the training data as well. Also, recomputing the system matrix factorization after each refinement or coarsening step, would again lead to a complexity of $\mathcal{O}(n^3)$, thus rendering the online step obsolete.

To maintain the improvements provided by the online-/offline splitting, matrix factorizations that allow to efficiently deal with spatial adaptivity have to be considered.

To begin with, let us investigate coarsening first. When removing any grid point i and its corresponding basis function φ_i, simply the i-th row and column have to be removed from the left-hand side $(R + \lambda C)$. Accordingly, adding grid points via refinement means extending the system matrix by additional rows and columns.

Regularization Another thing which comes to mind is the regularization term λC, that also is part of the left-hand side of the system (2.27). In many settings, for example when performing cross-validation, it would be desirable to vary the regularization strength λ during the online step as well. Thus, any matrix factorization that is used in the offline step should also allow for an efficient computation of adding and subtracting any multiple of the identity matrix from the left-hand side of the system. Note that for simplicity, we assumed the regularization operator C to be the Identity I.

Matrix Factorization Four different kinds of matrix factorization methods will be discussed and examined with respect to the two requirements mentioned before. An attempt that is quite easy to compute, would be LU factorization, where $R + \lambda C$ is decomposed into an lower triangular matrix L and upper triangular matrix U. As Figure 2.1 shows however, neither for spatial adaptivity nor for changes to the regularization strength, efficients methods are available.

Another approach is factorizing the left-hand side using Eigen decomposition, since it makes adapting the regularization strength λ quite simple. Because spatial adaptivity is still not supported by this factorization type, other methods have yet to be considered.

More promising seems to be the Cholesky decomposition, where the goal is to find a matrix L, such that the left-hand side can be decomposed into:

$$R + \lambda C = LL' \tag{2.28}$$

Matrix Factorization	Adaptivity	Regularization
LU decomposition	✗	✗
Eigen decomposition	✗	✓
Cholesky decomposition	✓	(✓)
Orthogonal decomposition	✓	✓

Table 2.1.: Examination of which matrix factorization method allows structural changes to the sparse grid (Adaptivity) and variation the regularization strength (Regularization).

As the system matrix is positive semidefinite, this approach is applicable. Updating the factorization with respect to refinement, coarsening and changes to the regularization strength were discussed and implemented by Adrian Sieler in [14]. One downside of this approach is that updating the factorization when changing the regularization strength λ has a computational complexity of $\mathcal{O}(n^3)$, making it as expensive as just recomputing the entire factorization from scratch. Therefore, theoretically the regularization strength can be adapted indeed, however not without significant computational effort that would render the entire Online-/Offline Splitting obsolete.

Another viable factorization method, the so-called Orthongal decomposition, was elaborated by Dimitrij Boschko in [3]. The system matrix factorizes into

$$R + \lambda C = QTQ' \tag{2.29}$$

where Q is an orthogonal and T an upper triangular matrix. As for the Cholesky decomposition, spatial adaptivity as well as varying the regularization strength are fully supported and also already implemented.

Figure 2.1 summarizes the capabilities of each matrix factorization approach with respect to the requirements that were discussed previously.

3. Implementation

3.1. The SG++ Toolbox

The SG++ toolbox is an open-source library, written in C++, which provides solutions to various problems using sparse grids. The project was created by Dirk Pflüger [11] and is currently developed at the chair of Scientific Computing at the Technical University of Munich and the Institute for Parallel and Distributed Systems at the University Stuttgart.

Among the many features this library implements, it also contains an approach to classification using sparse grid density estimation as discussed in Section 2.3. Previously however, this approach has not been part of the datamining pipeline, a module of SG++, that was designed to tackle data-driven problems in a very user-friendly fashion. Over the course of this thesis, the current implementation will be described and successively integrated in the pipeline. This requires refactoring of existing algorithmic components as well as implementation of new modules to fit the pipeline interface.

3.2. System matrix decomposition database

Since the offline step as described in Section 2.4 requires decomposing the system matrix, which in the case of Cholesky- and Orthogonal decomposition has cubic complexity, pre-computing those factorizations for common settings may help to improve the runtime of the sparse grid density estimation drastically. When encountering a scenario for which the left-hand side of the system (2.27) has already been decomposed in the past, the expensive offline step can be omitted, thus reducing the complexity of the entire density estimation to $\mathcal{O}(n^2)$, as it will only rely on the online step.

Of course, some kind of system to maintain these factorizations has to be introduced, in order to be able to benefit from this method. The first issue that arises is, how to formally identify similar scenarios, i.e. settings which share a system matrix. Simply comparing the entire left-hand side system matrix of each setting would certainly be one possible approach, however by far not the most efficient.

3.2.1. Configuration Structures

In the previous implementation, system matrices were built using a set of configuration structures, which therefore can be expected to be sufficient for the identification of possible application settings. Thus, the database will use an instantiation of these structures as a key value to identify certain scenarios.

GeneralGridConfiguration This configuration describes the basic structure of the sparse grid that will be created. It includes attributes such as the dimensionality, grid level and the type of basis function, the grid employs, as well as their behavior at the grid boundaries. Details can be found in [11].

AdaptivityConfiguration In this structure, the adaptivity behavior of the sparse grid can be defined, i.e. how many points should be added simultaneously during each refinement step. Also, adjustments to the heuristics, that are used for triggering refinements in the first place as well as for finding grid points suitable for refinement and coarsening respectively, are possible.

RegularizationConfiguration Settings regarding the regularization such as the regularization strength and the regularization operator are provided by this structure.

DensityEstimationConfiguration Lastly, the hyperparameters of the density estimation itself are included as members of this structure. Among those are the type of matrix factorization to employ, together with specific parameters that further specify the factorization behavior. Additionally, this configuration provides a member to decide whether to use an online-/offline splitting based approach to solve the system of linear equations or instead retrieve the solution numerically, for example by employing the conjugate gradients method.

3.2.2. Database Implementation

The database itself is represented by a JSON file. This allows any user to manually include decompositions themselves, if desired, but also makes the parsing process quite simple. The most relevant functionality, the database class provides, is to check whether a certain configuration is already associated with any factorized system matrix, implemented by the *hasDataMatrix* method. Consequently, if this is the case, a path to this serialized instance can be retrieved via the *getDataMatrix* method. Lastly, any setting of configurations can also be associated with a new file path by calling the *putDataMatrix* method. The class diagram given in Figure 3.1 depicts the structure of the database.

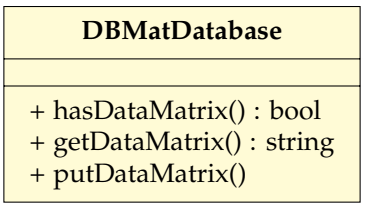


Figure 3.1.: Simplified class diagram of the system matrix decomposition database only including relevant methods. It provides functionality for retrieving and storing system matrices by associating a set of configurations with a path to the corresponding serialized and decomposed system matrix.

3.3. The Datamining Pipeline

As indicated by Figure 3.2, the datamining pipeline consists of various modules, that each provide different functionality to the concept. For an end-user, the only relevant component however is the *SparseGridMiner* class, as it encapsulates the entire datamining pipeline concept by its own. It can be easily instantiated using a factory class and passing a configuration file in JSON format. Currently, the *SparseGridMiner* implements datamining functionality, employing three different module classes:

DataSource The *DataSource* module is responsible for providing data samples in an iterative fashion. Files can automatically be processed and the dataset is split into batches of arbitrary size.

ModelFittingBase The *ModelFittingBase* base class represents the core of the functionality, as it implements the possible datamining approaches themselves. Depending on the specific subclass, the data samples retrieved from the *DataSource* instance can be used to train a model and also perform spatial refinement. Models for regression, density estimation as well as classification all inherit from this superclass.

Scorer The *Scorer* class uses a metric, that was specified in the configuration JSON file, in order to evaluate the performance of the model on the training and test data. Previously, it was responsible for triggering the fitting and spatial refinement by itself. This functionality however was moved into the *SparseGridMiner* class to ensure a more clean and extensible interface for the datamining pipeline.

3.4. Algorithm-/Application Refactoring

One of the main issues when migrating the density estimation and density estimation-based classification into the datamining pipeline was to reorganize ownership of different state instances. These instances include the grid itself as well as the vector of surpluses $\vec{\alpha}$ and the configurations described in Section 3.2.1. Those are meant to be held by the model instance of the pipeline, but were instead owned by the data matrix instances, see Figure 3.2. In the following, classes that relate to the left-hand side of the system matrix will be referred to as part of the algorithm module.

DBMatOffline The *DBMatOffline* class resembles a system matrix that may already be decomposed. It contains methods to build the system matrix based on a given set of configurations as well as functionality to factorize it. As especially for different factorization approaches the implementation of latter might differ fundamentally, the *DBMatOffline* is conceptualized as an abstract class type. Classes that inherit from it are:

- *DBMatOfflineLU*
- *DBMatOfflineChol*
- *DBMatOfflineDenseIChol*
- *DBMatOfflineEigen*
- *DBMatOfflineOrthoAdapt*

Each of them corresponds to a factorization method. The *DBMatOfflineDenseIChol* class represents an iterative approach to the Cholesky factorization, details can be found in [8].

In the previous implementation, each of those classes held the configuration attributes described in Section 3.2.1 themself, and used them to automatically create the underlying sparse grid, when the system matrix was requested to be built (by calling the *buildMatrix* method). As Figure 3.2 suggests however, the grid is to be owned by the model class of the pipeline instead. The same applies to the configuration instances as well. Thus, the *DBMatOffline* classes and its methods had to be refactored in a way, such that they can operate on a grid instance that is passed by the caller, instead of storing one themselves. Again, similar changes were necessary with regard to the configuration structures.

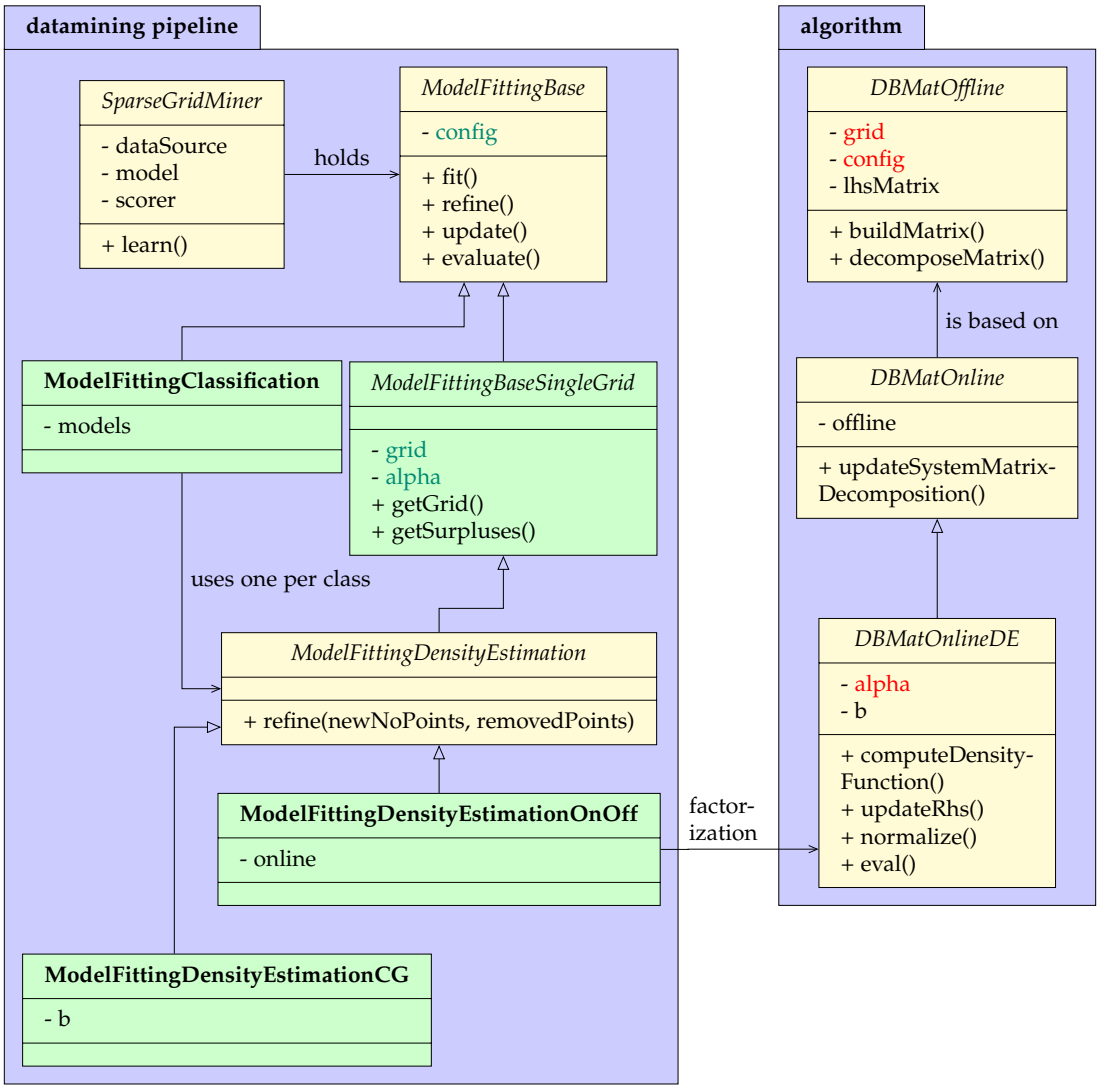


Figure 3.2.: Simplified class diagram of the datamining pipeline and an exert of the most relevant base classes of the algorithm modules. State instances were removed from the algorithm classes (red text) and placed inside the datamining pipeline structure (green text). Classes that had to be added are filled green, while already existing classes share a yellow filling.

DBMatOnline The *DBMatOnline* abstract class type is supposed to wrap around an instance of *DBMatOffline*, on which the system matrix decomposition already has been performed. It provides methods to update the left-hand side, i.e. performing refinement, coarsening and incorporating changes to the regularization strength λ.

DBMatOnlineDE The density estimation functionality itself is implemented into the *DBMatOnlineDE*, which itself inherits from the *DBMatOnline* class. There is one *DBMatOnline* subclass, each corresponding to a *DBMatOffline* class, which implements functionality tailored to fit the factorization it represents. Thus the following classes inherit from *DBMatOnlineDE*:

- *DBMatOnlineDELU*
- *DBMatOnlineDEChol*
- *DBMatOnlineDEDenseIChol*
- *DBMatOnlineDEEigen*
- *DBMatOnlineDEOrthoAdapt*

Each of those classes is related to a solver, that is supposed to compute the vector of surpluses $\vec{\alpha}$ for a given grid setting and the currently available right-hand side. Also, normalization functionality is implemented, ensuring that the density function, implicitly given by the surplus vector $\vec{\alpha}$, represents a valid probability distribution.

3.5. Integration of Density Estimation into the Datamining Pipeline

To implement the sparse grid density estimation into the datamining pipeline, a model class that inherits from *ModelFittingBase* had to be designed. Since density estimation, in contrast to classification (which relies on multiple density estimation models), only employs one sparse grid, the module hierarchy of the fitting module was split: The *ModelFittingBaseSingleGrid* class was created as another abstract supertype, which itself inherits from *ModelFittingBase*. It should serve as parent to all models, that only use a single sparse grid, i.e. density estimation and regression. This structure can be observed in Figure 3.2.

ModelFittingDensityEstimationOnOff There are two approaches to sparse grid density estimation, namely using the Online-/Offline Splitting as explained in Section 2.4 and employing iterative methods, such as conjugate gradients, to solve the system of linear equations. Consequentially, two different model classes were implemented, each corresponding to one of the two methods. The *ModelFittingDensityEstimationOnOff* class uses matrix factorization in order to retrieve the vector of surpluses $\vec{\alpha}$. Therefore, it refers to an instance of the *DBMatOnlineDE* class type, which is constructed when data is initially fit to the model using its *fit* method. Furthermore, whenever new data becomes available (for example in streaming settings or when performing learning on batches), the *update* method can be used to make the data known to the online object, i.e. recomputing the right-hand side of the equation and retrieving the corresponding new surpluses.

ModelFittingDensityEstimationCG Additionally, a class to perform density estimation employing the conjugate gradients method for solving the system of linear equations, given by Equation (2.27), was implemented as well. The functionality is very similar to the *ModelFittingDensityEstimationOnOff* class. In contrast to it however, the right-hand side is no longer held by an online instance but instead managed by the model itself.

3.6. Integration of Classification into the Datamining Pipeline

The classification module differs from other subclasses of *ModelFittingBase*, as it does not control a single grid and vector of surpluses, but instead uses references to several density estimation models, each of which is responsible for its own state instances. This structure can again be observed in Figure 3.2. As in order to perform density based estimation, a density function has to be computed for each class, the *ModelFittingClassification* module first splits the training data according to their class labels and then passes these datasets to the respective density estimation models.

When predicting the class of a data sample, the classification module uses the density estimation models it refers to, in order to evaluate all normalized class conditional density functions. Making use of Equation (2.8), the predicted class corresponds to the density function, that provides the highest evaluation on the data sample in question after being weighted with the respective class prior. This procedure is visualized by Figure 3.3.

One issue that naturally arises with this design is spatial adaptivity. Density estimation models, each of which inherits from the *ModelFittingDensityEstimation* superclass, select candidate points for refinement only based on the structure of their own grid or

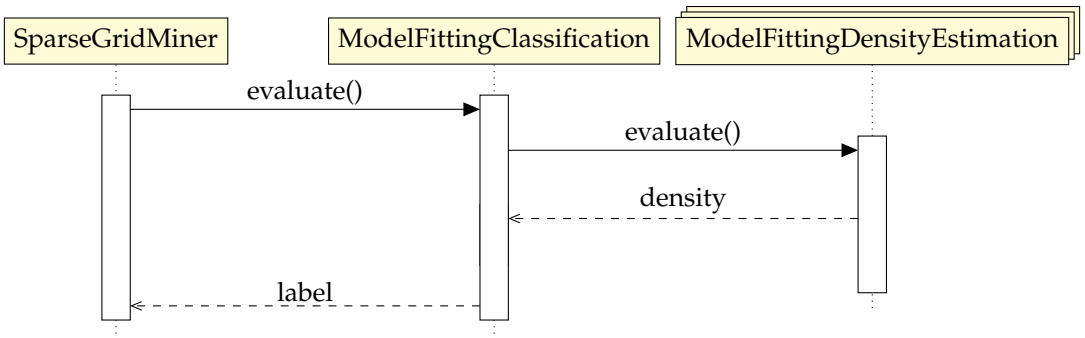


Figure 3.3.: Visualization of the classification procedure: The classification model uses different density estimation models and returns the class label associated with the density function, that provided the highest evaluation.

the magnitude of the their own surpluses. In classification however, more sophisticated methods are required: As elaborated by Kreisel in [7], the state of all models should be considered when selecting suitable points for refinement.

In the SG++ toolbox, refinement and coarsening is implemented by functors. Since classification relies on multiple grids at the same time, functors that operate on a set of models have to considered. Those inherit from the abstract superclass *MultiGridRefinementFunctor*. In especial, the classification model class supports these different types of refinement functors:

- *MultiSurplusRefinementFunctor*
- *ZeroCrossingRefinementFunctor*
- *DataBasedRefinementFunctor*
- *GridPointBasedRefinementFunctor*
- *MultipleClassRefinementFunctor*

For detailed explanations of those approaches, refer to [7]. Most of them however have in common, that they determine critical regions, i.e. where refinement might be necessary, by comparing the class conditional density functions with one another. Thus, state instances of all models, such as the underlying grids and corresponding surpluses, have to be accessible to the classification model. Because of that, models based on a single grid also offer respective getter-methods, as can be seen in Figure 3.2.

Since the functor instances automatically reorganize the grid instances as required for the refinement, changes have to be communicated back to the density models as

well. By overloading the *refine* method of the *ModelFittingDensityEstimation* class, each model can restructure its state instances based on an refinement step that was triggered externally and has already been applied to the grid. Figure 3.4 offers an illustration of this procedure.

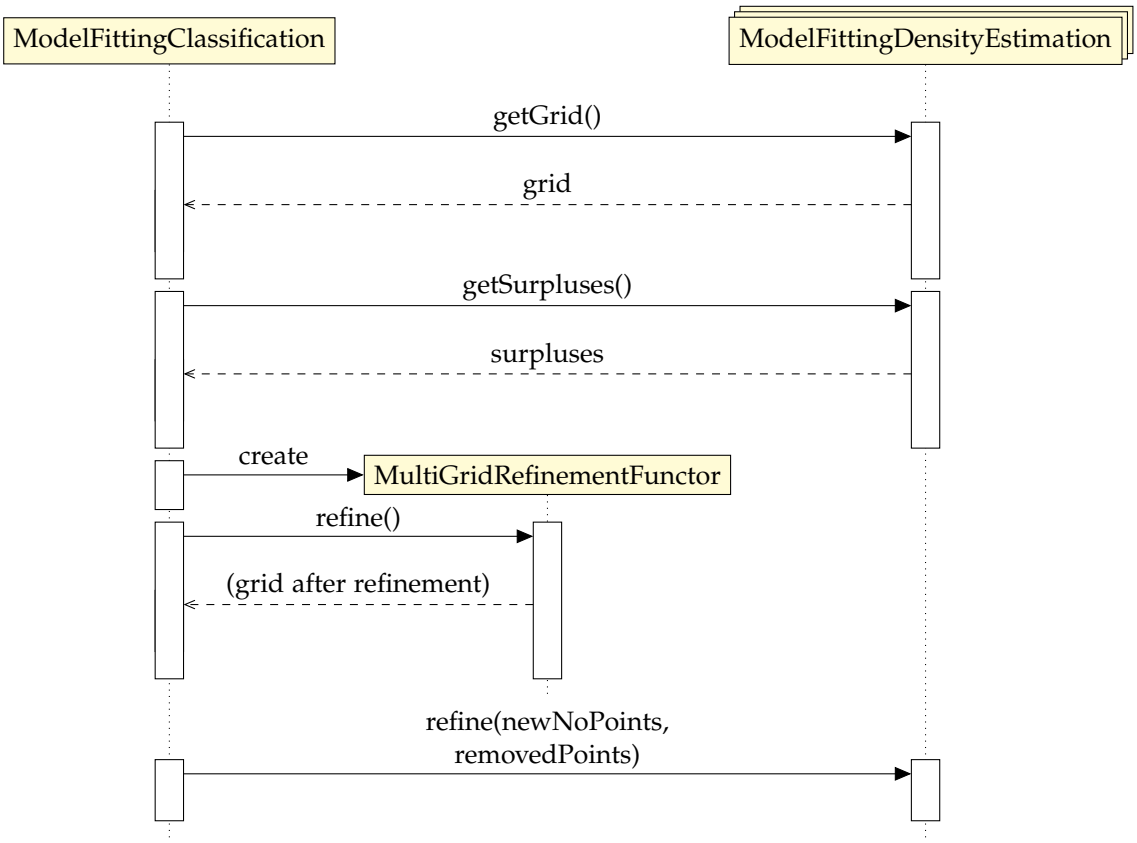


Figure 3.4.: Sequence diagram of the refinement procedure employing multiple grids. First, the classification model collects the state instances from each model. After that, a refinement functor is initialized and used to refine these grids. Lastly, the information about the changes in the grid structure is communicated back towards the density model.

4. Evaluation

In order to verify the implementation, its performance will be evaluated on different datasets. These measurements include the classification accuracy achieved by the models, as well as the runtime of the training process and overall memory consumption. The results will then be compared to the previous implementation of the approaches, as for each dataset density estimation based on Online-/Offline Splitting together with models employing conjugate gradients will be tested. All evaluations were performed on the same machine, using an *Intel Core i5-4210H* processor.

If not mentioned otherwise, the same configuration is used to train models on each dataset: We start with a regular sparse grid and vary the initial level between two and eight, which in the end results in sparse grids of different sizes. The training data is shuffled randomly and 20% of it will be used for validation, i.e. to monitor convergence of the training process and trigger refinements. The models for both implementations are trained for two epochs on batches of size ten in the case of Online-/Offline Splitting based density estimation and batches of size one when using conjugate gradients. Note that the previous implementation of conjugate gradients-based density estimation only supports batches of size one, that is, it processes each data sample separately. Thus, in order for the implementations to be comparable, the corresponding model of the datamining pipeline was configured to operate on a batch size of one as well. Ten refinements are performed, each refining ten grid points at once. A data-based strategy serves as refinement indicator. The regularization strength λ is set to 0.01 and the prior on the classes is obtained by the relative frequency of data samples belonging to a class.

4.1. The Ripley-Garcke Dataset

The Ripley-Garcke dataset originates from [12] and was modified by Jochen Garcke for being applicable to sparse grid methods. It was generated sampling noisily from a mixture of two Gaussians. It only contains a very small amount of samples, i.e. the training set consists of 250 points, while the testing set includes 1000 data instances. Since the Ripley-Garcke dataset is not linearly separable, it serves well for validating the classification functionality of the model implementation. A visualization of the dataset can be found in Figure 4.1.

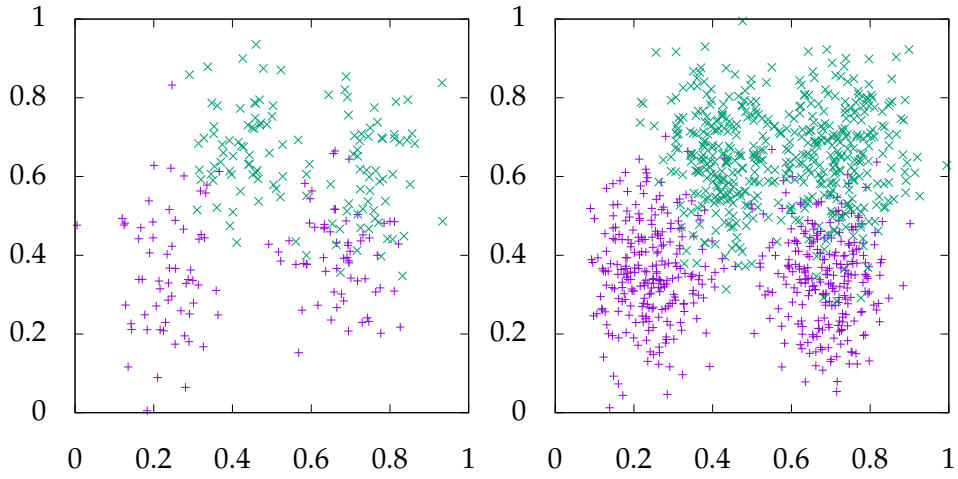


Figure 4.1.: Visualization of the Ripley-Garcke dataset. The training data (left) consists of 125 samples per class, while the testing data (right) includes 500 samples for each class.

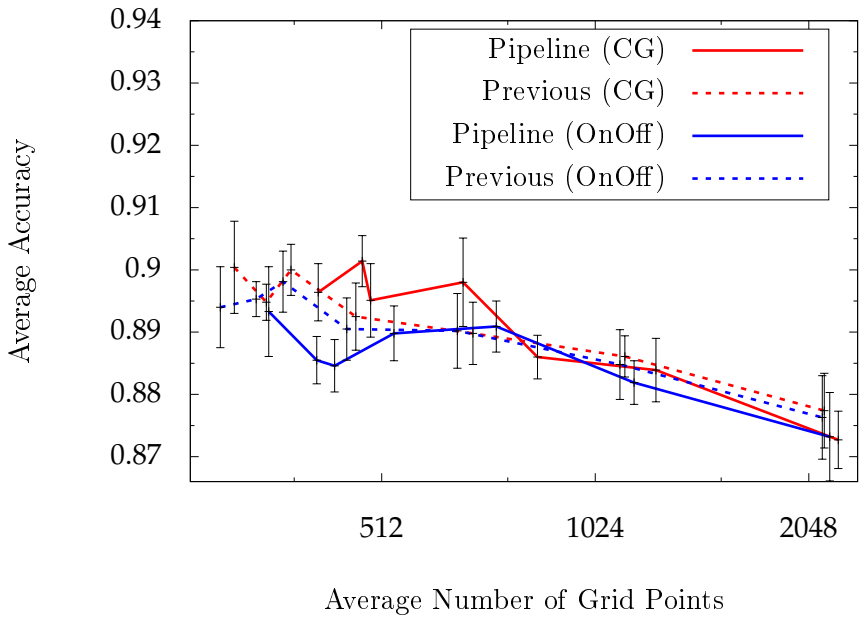


Figure 4.2.: Evaluation of the classification accuracy of the methods on the Ripley-Garcke dataset. Online-/Offline Splitting as well as Conjugate gradients-based approaches inside the datamining pipeline were compared to their previous implementation. Apart from small deviations, the performance of the pipeline and previous implementation mostly match. After reaching a certain grid size, the accuracy degrades slowly, probably because of overfitting.

4.1.1. Classification Accuracy

The first measurement to consider is the classification accuracy achieved by the different models. When increasing the initial sparse grid level we obtain different classifiers each relying on a more refined grid structure. After training the models, their classification accuracy on the test data is evaluated. The results are averaged over ten runs, while the corresponding standard deviation serves as uncertainty measure.

As figure 4.2 indicates, the performance of the pipeline model matches its previous implementation mostly. While the Online-/Offline Splitting-based method implemented into the datamining pipeline seems to slightly underperform, the approach employing conjugate gradients appears to achieve better results than its previous implementation. Note however, that these results suffer from uncertainty, as can also be observed in Figure 4.2. Nonetheless, the overall capability of the two implementations

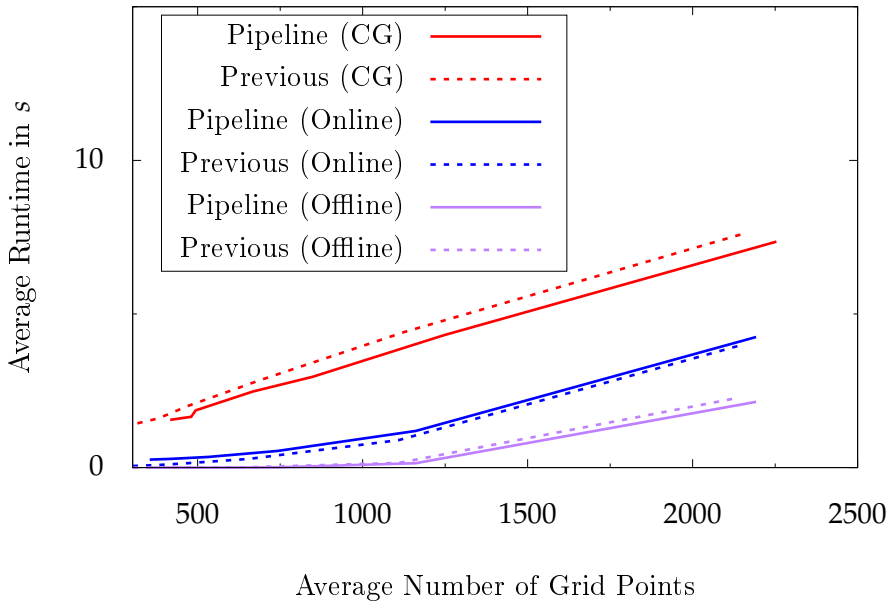


Figure 4.3.: Evaluation of the runtime of the methods on the Ripley-Garcke dataset. Online-/Offline Splitting as well as conjugate gradients-based approaches inside the datamining pipeline were compared to their previous implementation. The pipeline implementation achieves comparable runtimes and in case of the conjugate gradients approach, it even seems to outperform the previous implementation slightly. Overall, Online-/Offline Splitting results in superior runtime behaviour compared to the conjugate gradients method, even more so, when omitting the offline step.

can be regarded as quite similar. After a certain number of grid points is reached, the performance slowly degrades for all model implementations, probably because of overfitting, since the regularization strength is kept fixed while the model complexity steadily increases.

4.1.2. Time Complexity

Naturally, another measurement of interest is the time needed to train a model. Again, this measure heavily depends on the model complexity, i.e. the grid size. As for the classification accuracy, results are averaged over ten runs. Since the variance of the measurements is neglectably small, it is omitted in Figure 4.3.

As shown by Figure 4.3, both implementations for each approach, namely Online-

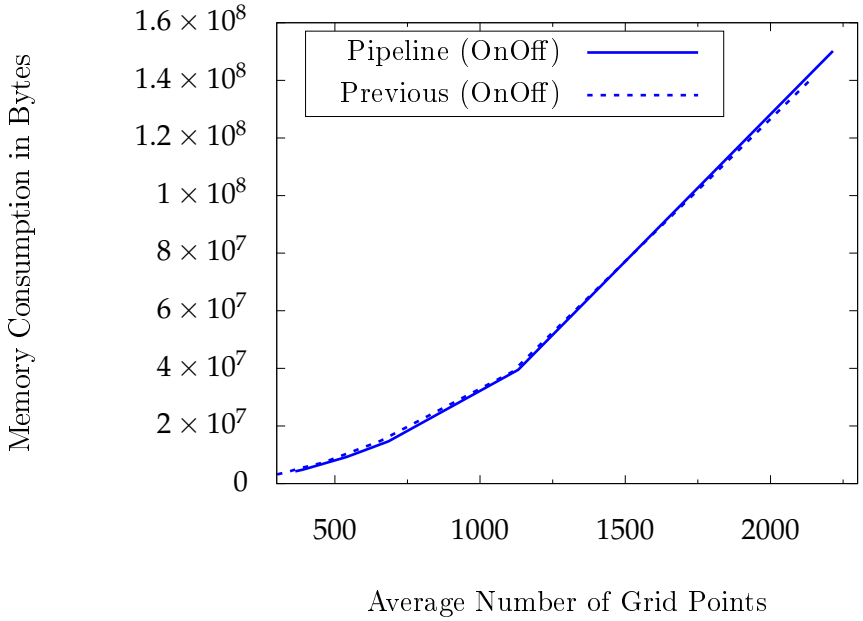


Figure 4.4.: Evaluation of the memory consumption of the Online-/Offline Splitting-based approach on the Ripley-Garcke dataset both inside the datamining pipeline and its previous implementation. The two implementations require almost the same amount of memory. Since the entire system matrix has to be present in memory during training, several megabytes are needed.

/Offline Splitting and conjugate gradients, almost exactly match in runtime behavior. The pipeline implementation of the conjugate gradients method even appears to outperform the previous implementation slightly. Generally, the Online-/Offline Splitting models can be observed to demand less computational effort when operating on the Ripley-Garcke dataset. This effect is even reinforced, when omitting the offline step, i.e. making use of the matrix factorization database mentioned in Section 3.2.

4.1.3. Memory Complexity

The third criterion that will be taken under consideration, is the memory consumed during training time, as this is another factor that puts limitations on the scalability of the model. Again, the memory complexity was evaluated for models that only differ by the size of the final grid they operate on. The exact values were retrieved making use of the valgrind [9] tool.

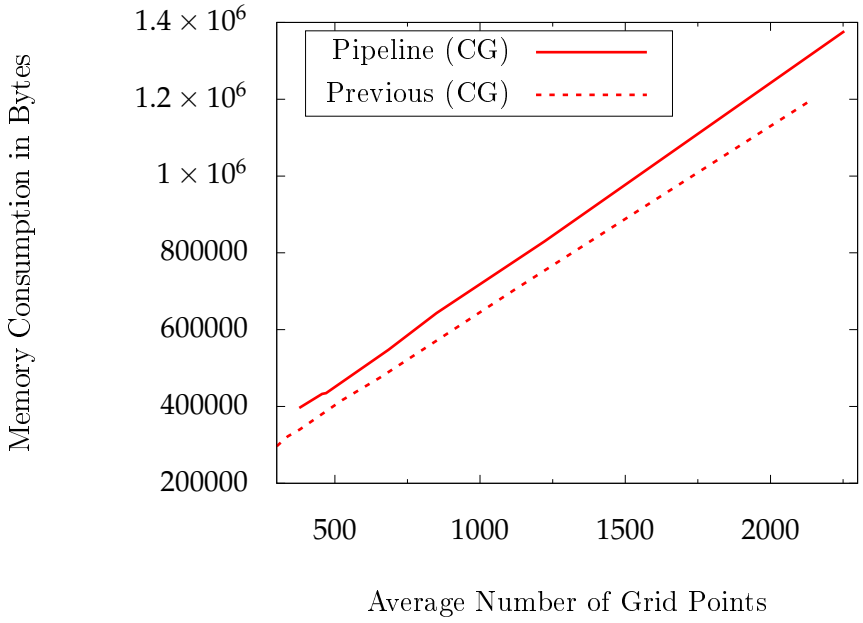


Figure 4.5.: Evaluation of the memory consumption of the conjugate gradients-based approach inside the datamining pipeline and its previous implementation on the Ripley-Garcke dataset. Even though the pipeline implementation results in a small overhead, both implementations only require memory in the order of kilobytes as the system matrix is not needed to be present in memory during training.

As illustrated by Figure 4.4, also the memory complexity of the two implementations match very well in the case of Online-/Offline Splitting-based density estimation. Similar results can be observed in Figure 4.5, where the overhead produced by the pipeline is comparably small and seems to be constant in the number of grid points. Another noteworthy aspect is, that the total amount of memory consumed by the Online-/Offline Splitting approach has a magnitude of megabytes, whereas the conjugate gradients method only requires few kilobytes during runtime. This is because, as mentioned in Section 2.3, the system matrix does not have be loaded into memory entirely during the solving process when using conjugate gradients [10].

4.2. The Two Moons Dataset

The second dataset that was used to evaluate the implementation is the Two Moons dataset, which also was generated artificially using Scipy [6]. The training data consists of 400 samples in total, while the testing data contains 600 data instances. All of them were sampled noisily with a standard deviation of 0.2. Figure 4.6 gives a visualization of the data.

4.2.1. Classification Accuracy

As for the Ripley-Garcke dataset, first of all the classification accuracy of the two implementations is evaluated on the Two Moons dataset. Again, models of different complexity are trained on the same data and evaluated on the test set afterwards, while the performance measurements were averaged over ten runs and the uncertainty is given by the empirical standard deviation.

Figure 4.7 shows that also for the Two Moons dataset the classification accuracy achieved by both the pipeline and the previous implementation does not differ significantly, when taking under consideration the uncertainty of the measurements. While the Online-/Offline Splitting-based approach results in a slightly worse accuracy than the previous implementation, pipeline models employing conjugate gradients are able to achieve small improvements in the classification accuracy. For larger grid sizes, the performance slowly starts to degrade eventually, probably because of overfitting, as has already been observed for the Ripley-Garcke dataset.

4.2.2. Time Complexity

Next, the time each model took to train was measured and compared with the previous implementation. As before, the grid size was varied by tweaking the initial sparse grid level, while all other settings remained fixed. Analogous to the Ripley-Garcke

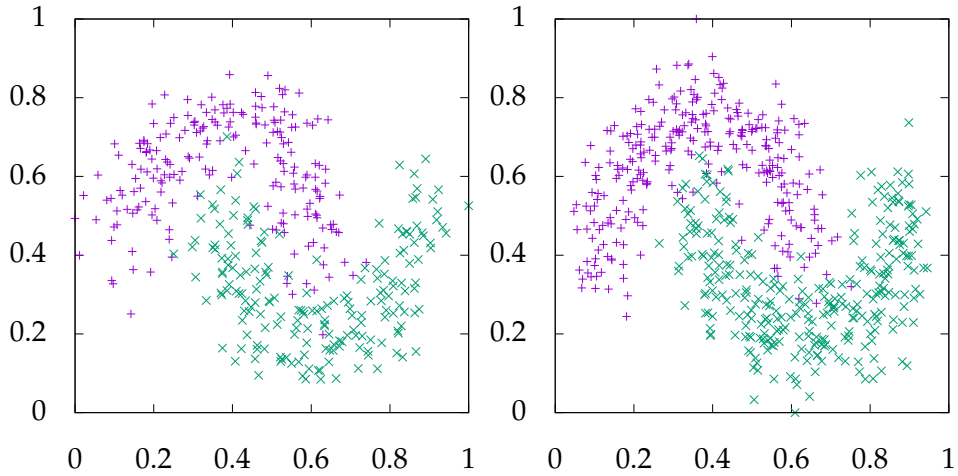


Figure 4.6.: Visualization of the Two Moons dataset. The training data (left) consists of 400 samples in total, while the testing data (right) includes 600 samples. For both sets, the samples were drawn noisily using Scipy [6] with a standard deviation of 0.2.

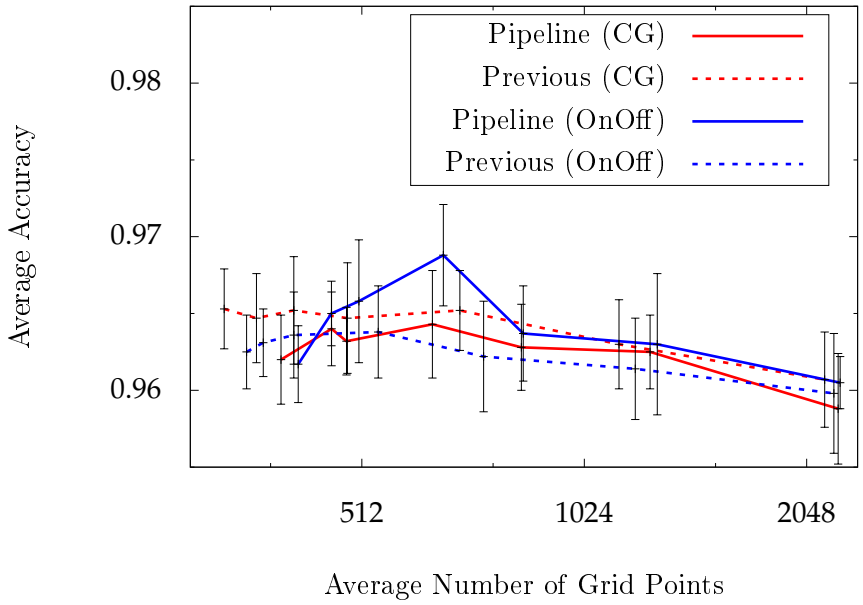


Figure 4.7.: Evaluation of the classification accuracy of the methods on the Two Moons dataset. Online-/Offline Splitting as well as conjugate gradients-based approaches inside the datamining pipeline were compared to their previous implementation. Within the given uncertainty, the pipeline implementation for both approaches achieves results similar to the previous implementation. While the Online-/Offline Splitting-based method slightly underperforms, pipeline models using conjugate gradients achieve small improvements in the classification accuracy.

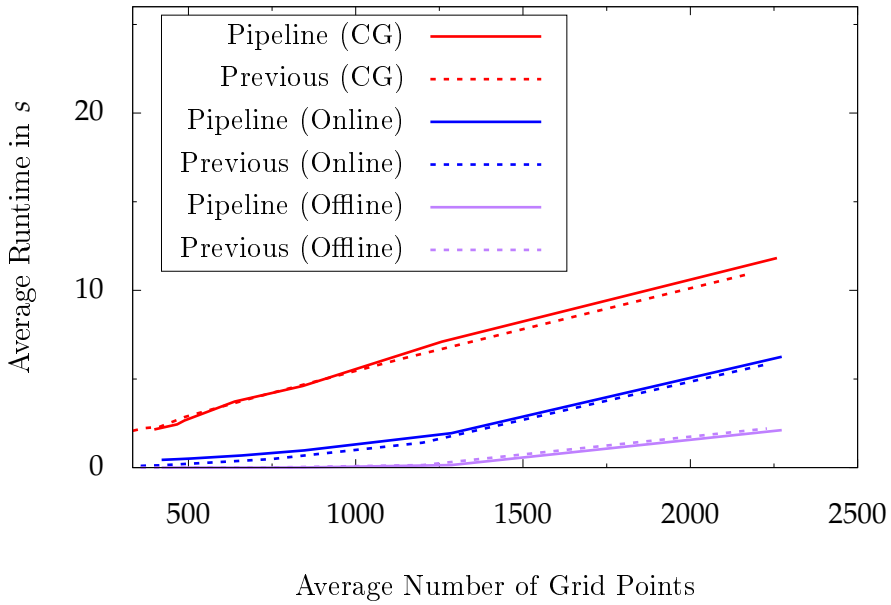


Figure 4.8.: Evaluation of the runtime of the methods on the Two Moons dataset. Both the pipeline as well as the previous implementation result in similar training times. Online-/Offline Splitting (both online- and offline step combined) slightly outperforms the conjugate gradients approach, while the online step alone requires significantly less training time.

dataset, measurements are averaged over ten runs and uncertainties are not included into Figure 4.8 because of their comparably small magnitude.

For the Two Moons dataset, Figure 4.8 shows, that the time complexity of the methods was indeed maintained when migrating the classification functionality into the datamining pipeline. For both Online-/Offline Splitting and the conjugate gradients-based approach, the runtime almost exactly matches the previous implementation. Also note that omitting the offline step (by using the matrix factorization database described in Section 3.2) results in a significant speedup of the training process.

4.2.3. Memory Complexity

Lastly, the memory requirements on the Two Moons dataset were analyzed. Models of different complexity were trained on the same dataset and the valgrind [9] tool was employed to retrieve memory consumption measurements.

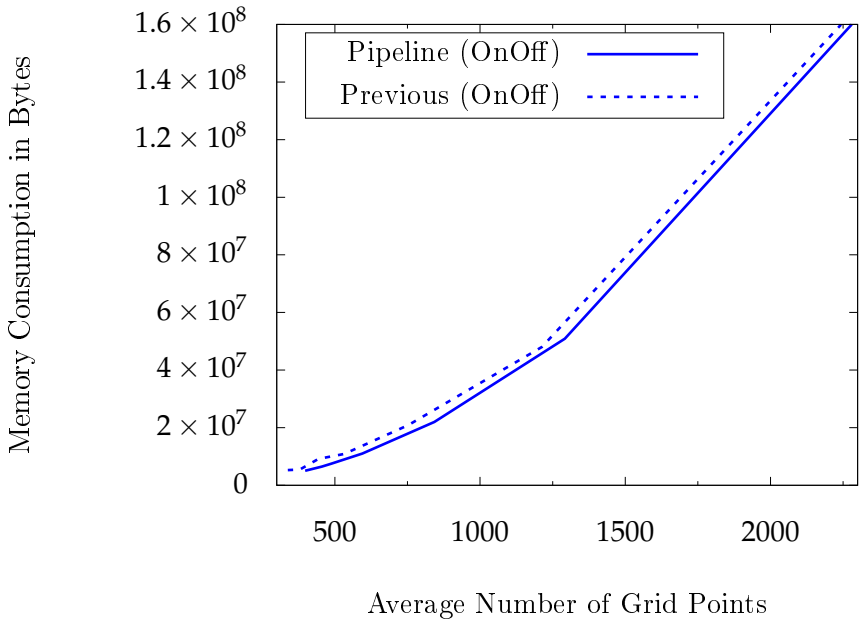


Figure 4.9.: Evaluation of the memory consumption of the Online-/Offline Splitting approach inside the datamining pipeline and its previous implementation on the Two Moons dataset. Both the pipeline implementation and the previous one require the same amount of memory. Since the entire system matrix has to be present in memory during the entire training process, the memory consumption has a magnitude of megabytes.

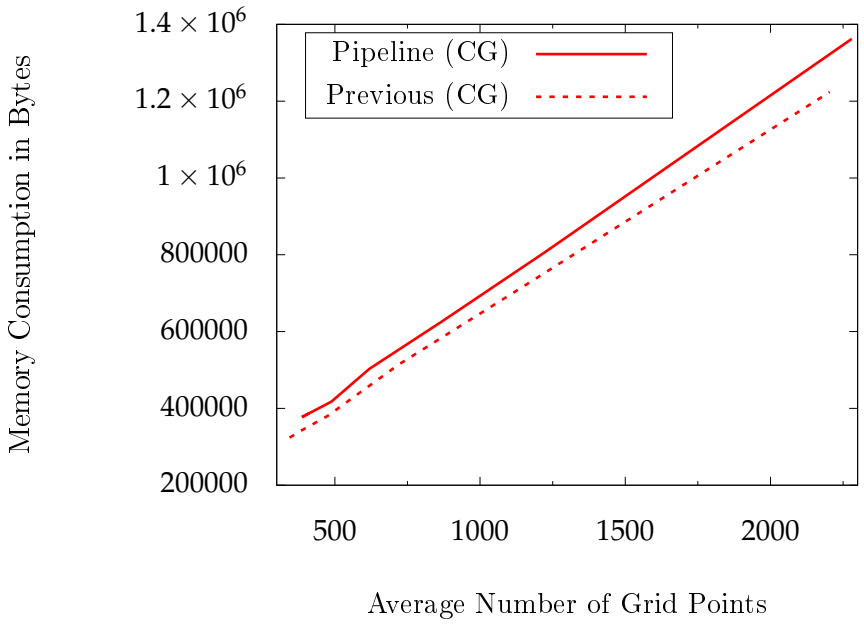


Figure 4.10.: Evaluation of the memory consumption of the conjugate gradients-based approach inside the datamining pipeline and its previous implementation on the Two Moons dataset. Both implementations require almost the same amount of memory during training. Again, the pipeline implementation causes a slight overhead of small magnitude. The total memory consumption has only an order of kilobytes, as it is not necessary to load the system matrix into memory entirely.

The behavior of the models on the Two Moons dataset, depicted by Figures 4.9 and 4.10, is very similar to the observations made for the Ripley-Garcke data: First, for both approaches the memory requirement curves of the pipeline and previous implementation match fairly well. The conjugate gradients-based method of the pipeline however still causes a slight memory overhead. This indicates, that the integration of the methods into the datamininig pipeline did not have any significant bad influence the memory complexity overall. Second, the matrix factorization approach in the pipeline as well as in the previous implementation again demand several megabytes of memory, as the entire system matrix has to be kept in memory during the whole training process [10].

4.3. The DR-10 Dataset

The third dataset that will be used to verify the models of the pipeline is the data release 10 of the Sloan Digital Sky Survey 3 (DR-10)[1] [13]. It contains several photometric measurements of astronomical objects, resulting in a four-dimensional dataset with over $640,000$ training and over $270,000$ testing samples. As the only non-artificial dataset of significant size used in this thesis, it provides insight on how the models behave on very large real world data. Using the DR-10 dataset, the batch size, the models were trained on, was increased to $50,000$ in order to make training more tractable. Since the previous implementation of conjugate gradients-based density estimation only supports batches of size one, it is not applicable to a big dataset such as the DR-10. Training even a single classifier operating on small grids took several hours. This is due to the fact, that the system of linear equations has to be solved again whenever any new data sample is processed. In this Section, the previous implementation of the conjugate gradients approach will therefore not be considered, while the conjugate gradients models of the datamining pipeline were also trained on batches of size $50,000$. Furthermore, the regularization strength was decreased to $\lambda = 10^{-7}$ and the initial sparse grid level was

[1] Official SDSS-III Acknowledgement: Funding for SDSS-III has been provided by the Alfred P. Sloan Foundation, the Participating Institutions, the National Science Foundation, and the U.S. Department of Energy Office of Science. The SDSS-III web site is http://www.sdss3.org/. SDSS-III is managed by the Astrophysical Research Consortium for the Participating Institutions of the SDSS-III Collaboration including the University of Arizona, the Brazilian Participation Group, Brookhaven National Laboratory, Carnegie Mellon University, University of Florida, the French Participation Group, the German Participation Group, Harvard University, the Instituto de Astrofisica de Canarias, the Michigan State/Notre Dame/JINA Participation Group, Johns Hopkins University, Lawrence Berkeley National Laboratory, Max Planck Institute for Astrophysics, Max Planck Institute for Extraterrestrial Physics, New Mexico State University, New York University, Ohio State University, Pennsylvania State University, University of Portsmouth, Princeton University, the Spanish Participation Group, University of Tokyo, University of Utah, Vanderbilt University, University of Virginia, University of Washington, and Yale University.

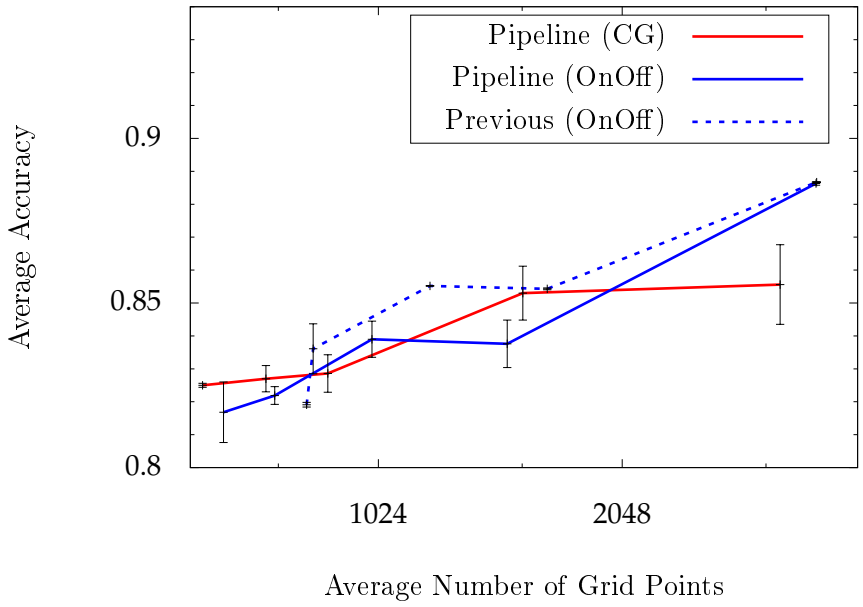


Figure 4.11.: Evaluation of the classification accuracy of the methods on the DR-10 dataset. Online-/Offline Splitting as well as conjugate gradients-based approaches inside the datamining pipeline were compared to their previous implementation. The Online-/Offline Splitting-based approach inside the pipeline is only able to catch up to its previous implementation on bigger grids. The conjugate gradients-based method achieves comparable results but fails to benefit from a big grid as the Online-/Offline Splitting density estimation does.

only varied between two and six.

4.3.1. Classification Accuracy

Analogously to the previous datasets, classifiers operating on different grid sizes were trained separately. Since training any model on the DR-10 dataset took significantly more time than using a small dataset, measurements for the classification accuracy were averaged over only three runs.

As can be seen in Figure 4.11, both implementations of Online-/Offline Splitting achieve comparable results. The previous implementation however slightly outperforms the pipeline model on smaller grid sizes. Only when operating on a higher number of grid points, the pipeline implementation is able to catch up to the previous state.

The conjugate gradients-based method also is able to provide reasonable results, but seems to not benefit from a bigger grid as much as the Online-/Offline splitting method does. More promising results can be expected when using even larger grids, since the accuracy curves do not degrade for more refined grid settings, in contrast to results that have been observed previously for the Ripley-Garcke- and Two Moons dataset.

4.3.2. Time Complexity

As for the other datasets, the runtime of the model training processes will be examined as well. Again, because of the size of the DR-10 dataset, results were averaged over only three runs. Uncertainties are once more left out in Figure 4.12, as they are neglectable, compared to the overall runtime.

Figure 4.12 mainly shows three things: First, the two approaches to density estimation implemented into the datamining pipeline result in comparable runtimes, while the online step of the Online-/Offline Splitting-based classifier slightly outperforms the conjugate gradients method. Second, the previous implementation of Online-/Offline splitting takes significantly longer to train a model. This issue is caused due to the previous implementation considering the entire training data when applying the data-based refinement strategy to the model, while the pipeline only has access to the current batch. The third noteworthy observation is, that the offline step of the Online-/Offline Splitting-based approach does almost not at all contribute to the overall runtime of the training process. That is, because the effort to decompose the system matrix for comparably small grid sizes is neglectable compared to processing batches consisting of $50,000$ data samples each.

4.3.3. Memory Complexity

Lastly, the memory complexity of the models will be evaluated on the DR-10 dataset. Models operating on different grid sizes were trained separately and memory consumption measurements were retrieved using the valgrind [9] tool.

The observations for the Online-/Offline approach, depicted by Figure 4.13, are similar to those made on the other datasets. While both implementations require about the same memory, the overall consumption is still dominated by the system matrix itself, even though the entire data has to be kept in memory during training as well. In the case of conjugate gradients-based density estimation however, the memory complexity seems to be constant, as can be seen in Figure 4.14. This is due to the fact, that the memory allocated for the actual training process itself is neglectably small compared to the amount of memory, which is required to hold the training and testing data. Thus, when operating on big datasets, for smaller grids, the memory complexity is dominated

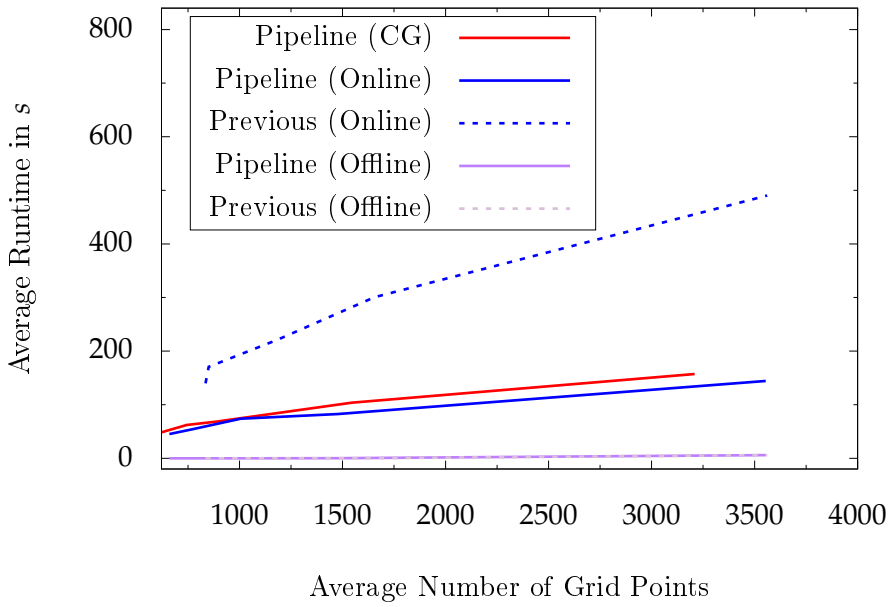


Figure 4.12.: Evaluation of the runtime of the methods on the DR-10 dataset. The Online-/Offline Splitting approach was compared to its previous implementation, while for the conjugate-gradients method only the pipeline implementation was considered. The two approaches to density estimation implemented into the datamining pipeline seem to be able to cope with big datasets better when using data-based refinement, as they only consider the current batch rather than the entire training data when triggering refinements. The online step also dominates the offline step by far, since the effort to process batches of size $50,000$ is way higher compared to factorizing the system matrix.

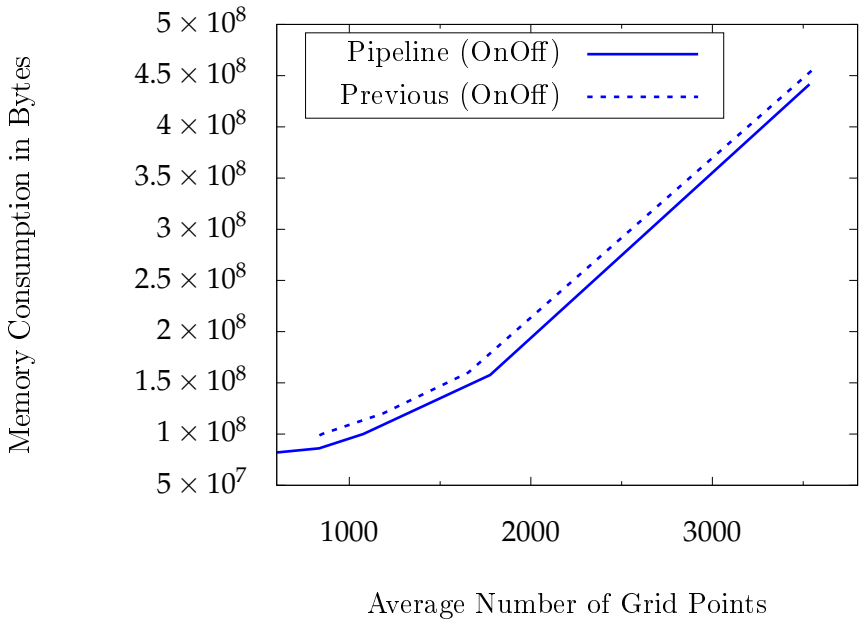


Figure 4.13.: Evaluation of the memory consumption of the Online-/Offline Splitting-based approach inside the datamining pipeline and its previous implementation on the DR-10 dataset. The memory consumption of the pipeline implementation is slightly reduced compared to the previous implementation. The overall memory complexity still is dominated by the size of the system matrix, even when operating on large datasets such as the DR-10.

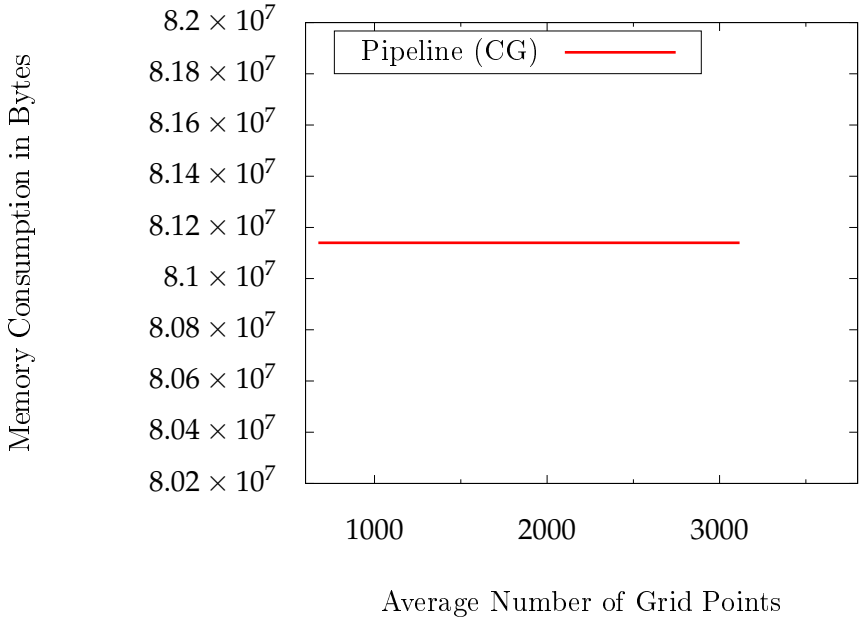


Figure 4.14.: Evaluation of the memory consumption of the conjugate gradients-based approach inside the datamining pipeline. Since the system matrix is not required to be loaded into memory entirely, the overall memory complexity is dominated by the size of the dataset.

by the size of the data itself rather than the number of points in the underlying sparse grids.

5. Conclusion and Future Work

Over the course of this thesis, density-estimation based classifiers operating on sparse grids have been successfully integrated into the datamining pipeline of the SG++ toolbox. This involved implementing suitable model classes into the pipeline as well as refactoring algorithmic modules, in order to fit the concept of the pipeline. State instances are now controlled by the model instances rather than the algorithmic components.

Additionally, density estimation models were made accessible to the end-user by implementing them as standalone modules into the dataminig pipeline as well. These include a model that relies on factorizing the system matrix in order to obtain the density function together with a conjugate gradients-based approach. To enhance the runtime performance of density estimation approaches employing matrix factorization, a database system to utilize precomputed factorizations for common scenarios has been introduced.

Evaluations have proven the models to maintain the performance of their previous implementation in terms of classification accuracy as well as time and memory complexity of the training process apart from neglectable overhead produced by the pipeline framework itself. For big datasets and data-based refinement strategies, the runtime behaviour of the methods could even be enhanced.

Further improvements to the classification models could be done by parallelizing the learning process inside the pipeline. Such concepts have already been elaborated and implemented previously by Vincent Bode in [2] and could be transfered to the classification model of the datamining pipeline.

A. Experimental Results

Avg. grid size	Avg. accuracy	Accuracy uncertainty	Avg. runtime online / offline (s)	Runtime uncertainty online / offline (s)
414.50	0.8855	0.0038	0.2801/ 0.0000	0.0119/ 0.0000
439.40	0.8846	0.0042	0.2939/ 0.0000	0.0185/ 0.0000
532.70	0.8898	0.0044	0.3488/ 0.0001	0.0204/ 0.0003
742.50	0.8909	0.0041	0.5443/ 0.0100	0.0273/ 0.0000
1161.40	0.8819	0.0035	1.1951/ 0.1444	0.0943/ 0.0054
2192.50	0.8732	0.0071	4.2503/ 2.1392	0.2073/ 0.0413

Table A.1.: Runtime and accuracy measurements of the pipeline implementation of Online-/Offline Splitting-based methods on the Ripley-Garcke dataset

Avg. grid size	Avg. accuracy	Accuracy uncertainty	Avg. runtime online / offline (s)	Runtime uncertainty online / offline (s)
340.54	0.8953	0.0028	0.0689/ 0.0000	0.0111/ 0.0000
371.63	0.8981	0.0049	0.0803/ 0.0000	0.0043/ 0.0000
457.01	0.8905	0.0050	0.1332/ 0.0000	0.0136/ 0.0000
654.42	0.8902	0.0060	0.2834/ 0.0112	0.0318/ 0.0013
1109.22	0.8848	0.0056	0.8930/ 0.1526	0.0610/ 0.0102
2139.51	0.8763	0.0067	3.9639/ 2.2733	0.4294/ 0.1499

Table A.2.: Runtime and accuracy measurements of the previous implementation of Online-/Offline Splitting-based methods on the Ripley-Garcke dataset

Avg. grid size	Avg. accuracy	Accuracy uncertainty	Avg. runtime (s)	Runtime uncertainty (s)
480.55	0.9014	0.0041	1.6555	0.1610
493.75	0.8951	0.0059	1.8634	0.1670
666.75	0.8980	0.0071	2.4738	0.2792
848.75	0.8860	0.0035	2.9567	0.3272
1247.10	0.8839	0.0051	4.3123	0.3677
2253.70	0.8727	0.0046	7.3554	0.5801

Table A.3.: Runtime and accuracy measurements of the pipeline implementation of conjugate gradient-based methods on the Ripley-Garcke dataset

Avg. grid size	Avg. accuracy	Accuracy uncertainty	Avg. runtime (s)	Runtime uncertainty (s)
352.00	0.8948	0.0029	1.5272	0.1966
381.30	0.9000	0.0041	1.6012	0.1664
470.60	0.8925	0.0054	2.0046	0.1853
688.50	0.8898	0.0050	2.8423	0.3798
1127.10	0.8861	0.0033	4.4174	0.3400
2153.60	0.8774	0.0060	7.6103	0.6684

Table A.4.: Runtime and accuracy measurements of the previous implementation of conjugate gradient-based methods on the Ripley-Garcke dataset

Avg. grid size	Avg. accuracy	Accuracy uncertainty	Avg. runtime online / offline (s)	Runtime uncertainty online / offline (s)
465.60	0.9650	0.0021	0.4749/ 0.0000	0.0161/ 0.0000
507.30	0.9658	0.0040	0.5086/ 0.0000	0.0175/ 0.0000
659.90	0.9688	0.0033	0.6840/ 0.0000	0.0460/ 0.0000
846.60	0.9637	0.0031	0.9681/ 0.0101	0.0407/ 0.0003
1285.60	0.9630	0.0046	1.9444/ 0.1406	0.0841/ 0.0043
2273.00	0.9605	0.0017	6.2499/ 2.1131	0.2157/ 0.0207

Table A.5.: Runtime and accuracy measurements of the pipeline implementation of Online-/Offline Splitting-based methods on the Two Moons dataset

Avg. grid size	Avg. accuracy	Accuracy uncertainty	Avg. runtime online / offline (s)	Runtime uncertainty online / offline (s)
376.57	0.9631	0.0022	0.1095/ 0.0000	0.0167/ 0.0000
414.11	0.9636	0.0028	0.1309/ 0.0000	0.0151/ 0.0000
538.86	0.9638	0.0030	0.2524/ 0.0000	0.0574/ 0.0000
748.25	0.9622	0.0036	0.4796/ 0.0105	0.0768/ 0.0011
1200.09	0.9614	0.0033	1.4053/ 0.1459	0.1590/ 0.0083
2228.94	0.9598	0.0039	5.8331/ 2.2086	0.8007/ 0.1869

Table A.6.: Runtime and accuracy measurements of the previous implementation of Online-/Offline Splitting-based methods on the Two Moons dataset

Avg. grid size	Avg. accuracy	Accuracy uncertainty	Avg. runtime (s)	Runtime uncertainty (s)
465.95	0.9640	0.0024	2.4360	0.1585
488.05	0.9632	0.0022	2.6533	0.1446
637.90	0.9643	0.0035	3.7241	0.3803
841.65	0.9628	0.0028	4.6031	0.3441
1257.25	0.9625	0.0024	7.1039	0.6387
2258.85	0.9588	0.0036	11.8249	0.6405

Table A.7.: Runtime and accuracy measurements of the pipeline implementation of conjugate gradients-based methods on the Two Moons dataset

Avg. grid size	Avg. accuracy	Accuracy uncertainty	Avg. runtime (s)	Runtime uncertainty (s)
368.70	0.9647	0.0029	2.2242	0.2973
414.20	0.9652	0.0035	2.3035	0.2255
489.50	0.9647	0.0036	2.8577	0.4055
695.00	0.9652	0.0026	3.9426	0.3955
1141.10	0.9630	0.0029	6.1578	0.4825
2165.80	0.9607	0.0031	10.8836	0.6818

Table A.8.: Runtime and accuracy measurements of the previous implementation of conjugate gradients-based methods on the Two Moons dataset.

Avg. grid size	Avg. accuracy	Accuracy uncertainty	Avg. runtime online / offline (s)	Runtime uncertainty online / offline (s)
762.67	0.8219	0.0027	53.4016/ 0.0000	4.0137/ 0.0000
1005.67	0.8390	0.0055	73.7837/ 0.0020	3.5992/ 0.0000
1477.33	0.8376	0.0072	82.6631/ 0.1433	5.5665/ 0.0049
3553.67	0.8863	0.0005	144.1860/ 5.9217	6.2422/ 0.0742

Table A.9.: Runtime and accuracy measurements of the pipeline implementation of Online-/Offline Splitting-based methods on the DR-10 dataset.

Avg. grid size	Avg. accuracy	Accuracy uncertainty	Avg. runtime online / offline (s)	Runtime uncertainty online / offline (s)
850.67	0.8361	0.0076	170.5783/ 0.0000	1.7757/ 0.0000
1186.00	0.8552	0.0001	221.4463/ 0.0020	0.7872/ 0.0000
1655.00	0.8543	0.0003	300.5573/ 0.1463	1.2353/ 0.0031
3561.00	0.8867	0.0000	490.3910/ 5.9860	2.6678/ 0.1217

Table A.10.: Runtime and accuracy measurements of the previous implementation of Online-/Offline Splitting-based methods on the DR-10 dataset.

Avg. grid size	Avg. accuracy	Accuracy uncertainty	Avg. runtime (s)	Runtime uncertainty (s)
744.00	0.8270	0.0040	62.1821	2.5060
887.00	0.8286	0.0057	68.4133	7.8965
1544.00	0.8530	0.0082	103.6683	3.3741
3208.67	0.8556	0.0121	157.2517	7.3188

Table A.11.: Runtime and accuracy measurements of the pipeline implementation of conjugate gradients-based methods on the DR-10 dataset.

Avg. grid size	Memory (Bytes)
387.00	4784000
402.00	5181000
541.00	9257000
685.00	14680000
1131.00	39480000
2215.00	150200000

Table A.12.: Memory measurements of the pipeline implementation of Online-/Offline Splitting-based methods on the Ripley-Garcke dataset.

Avg. grid size	Memory (Bytes)
337.80	4091000
373.00	5132000
460.60	7295000
656.10	14750000
1111.80	38840000
2135.40	140000000

Table A.13.: Memory measurements of the previous implementation of Online-/Offline Splitting-based methods on the Ripley-Garcke dataset.

Avg. grid size	Memory (Bytes)
456.00	432600
470.00	434700
685.00	547500
850.00	643100
1222.00	830100
2254.00	1377000

Table A.14.: Memory measurements of the pipeline implementation of conjugate gradients-based methods on the Ripley-Garcke dataset.

Avg. grid size	Memory (Bytes)
338.00	322100
379.00	340100
525.00	416500
656.00	475500
1118.00	703500
2139.00	1197000

Table A.15.: Memory measurements of the previous implementation of conjugate gradients-based methods on the Ripley-Garcke dataset.

Avg. grid size	Memory (Bytes)
455.00	6487000
488.00	7524000
595.00	11010000
843.00	22000000
1292.00	50880000
2324.00	164900000

Table A.16.: Memory measurements of the pipeline implementation of Online-/Offline Splitting-based methods on the Two Moons dataset.

Avg. grid size	Memory (Bytes)
379.40	5465000
443.90	9187000
532.40	10890000
752.90	20900000
1212.80	48040000
2242.70	159900000

Table A.17.: Memory measurements of the previous implementation of Online-/Offline Splitting-based methods on the Two Moons dataset.

Avg. grid size	Memory (Bytes)
387.00	377400
488.00	417300
618.50	502900
869.00	626500
1222.00	806200
2280.00	1362000

Table A.18.: Memory measurements of the pipeline implementation of conjugate gradients-based methods on the Two Moons dataset.

Avg. grid size	Memory (Bytes)
355.00	329000
374.00	337800
475.00	380300
720.00	514500
1142.00	714000
2204.00	1224000

Table A.19.: Memory measurements of the previous implementation of conjugate gradients-based methods on the Two Moons dataset.

Avg. grid size	Memory (Bytes)
834.00	86140000
1076.00	100000000
1776.00	157700000
3535.00	441500000

Table A.20.: Memory measurements of the pipeline implementation of Online-/Offline Splitting-based methods on the DR-10 dataset.

Avg. grid size	Memory (Bytes)
851.00	100100000
1186.00	120200000
1655.00	160100000
3561.00	457200000

Table A.21.: Memory measurements of the previous implementation of Online-/Offline Splitting-based methods on the DR-10 dataset.

Avg. grid size	Memory (Bytes)
779.00	81140000
787.00	81140000
1338.00	81140000
3116.00	81140000

Table A.22.: Memory measurements of the pipeline implementation of conjugate gradients-based methods on the DR-10 dataset.

Bibliography

[1] C. M. Bishop. Pattern Recognition and Machine Learning (Information Science and Statistics). Berlin, Heidelberg: Springer-Verlag, 2006. ISBN: 0387310738.

[2] V. Bode. Parallelization of a Sparse Grids Batch Classifier. Bachelor's thesis. Department of Informatics, Technical University of Munich, Sept. 2017.

[3] D. Boschko. Orthogonal Matrix Decomposition for Adaptive Sparse Grid Density Estimation Methods. Bachelor's thesis. Technical University of Munich, Sept. 2017.

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

[5] T. Hastie, R. Tibshirani, and J. Friedman. The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition. Springer Series in Statistics. Springer New York, 2009. ISBN: 9780387848587.

[6] E. Jones, T. Oliphant, P. Peterson, et al. *SciPy: Open source scientific tools for Python*. Accessed Aug. 2018. 2001–.

[7] S. Kreisel. Spatial Refinement for Sparse Grid Classifiers. Bachelor's thesis. Department of Informatics, Technical University of Munich, June 2016.

[8] M. Lettrich. Iterative Incomplete Cholesky Decomposition for Datamining using Sparse Grids. Studienarbeit/SEP/IDP. Department of Mathematics, May 2017.

[9] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In: *SIGPLAN Not.* 42.6 (June 2007), pp. 89–100. ISSN: 0362-1340. DOI: 10.1145/1273442.1250746.

[10] B. Peherstorfer. Model Order Reduction of Parametrized Systems with Sparse Grid Learning Techniques. Dissertation. Department of Informatics, Technical University of Munich, Oct. 2013.

[11] D. Pflüger, B. Peherstorfer, and H.-J. Bungartz. Spatially adaptive sparse grids for high-dimensional data-driven problems. In: *Journal of Complexity* 26.5 (Oct. 2010). published online April 2010, pp. 508–522. ISSN: 0885-064X.

[12] B. D. Ripley and N. L. Hjort. Pattern Recognition and Neural Networks. 1st. New York, NY, USA: Cambridge University Press, 1995. ISBN: 0521460867.

[13] *SDSS DR-10*. `https://www.sdss3.org/dr10/`,Accessed Aug. 2018.

[14] A. Sieler. Refinement and Coarsening of Online-Offline Data Mining Methods with Sparse Grids. Bachelor's thesis. Department of Mathematics, Technical University of Munich, Mar. 2016.