DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Combigrid Based Dimensional Adaptivity for Sparse Grid Density Estimation and Classification

Nico Rösel

DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Combigrid Based Dimensional Adaptivity for Sparse Grid Density Estimation and Classification

Combigitter-basierte Dimensionsadaptivität für Dünngitter Dichteschätzung und Klassifikation

Author:	Nico Rösel
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	Kilian Röhner, M.Sc.
	Michael Obersteiner, M.Sc.
Submission Date:	15.04.2019

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

Munich, 15.04.2019 Nico Rösel

Acknowledgments

I would like to thank Michael and Kilian for their advise and for keeping the weekly meetings fun.
I would like to thank those of my friends who made it to lunch in time for not letting me wait and those who did not for paying my coffee.
I would like to thank my family and Matilde for the proofreading and for keeping me motivated.

Abstract

In the frame of this thesis the sparse grid combination technique was used to expand the density estimation based approach to classification in the SG++ framework. Thanks to the independence of the component grids, it was possible to implement much faster dimensional adaptive refinements. While the old implementation had to refit the whole grid after each refinement step, now just the additionally added components have to be fitted. The thesis contains the theoretical background, a description, and an evaluation of the implementation.

Contents

1. Introduction

Over the last decades computers gained a tremendous amount of power. There is an ever-growing number of tasks in which computers outperform humans. Their greatest deficiency is that they need a precise set of instructions for every thing they do. This set of instructions is called a computer program and, depending on the problem, defining it can get very costly. But there are techniques that allow to train a program from experience or sample data instead of manually coding its whole behavior. These techniques are summarized under the term machine learning.
A very common machine learning task is classification, i.e. identifying a sample as an instance of a class. Classification has a broad spectrum of applications. Photo apps use it to sort pictures according to the persons in it, spam filters use it to classify incoming data into spam and non-spam content, and in medicine it is even used to distinguish between cancer and normal cells.
There are different ways to make a classification decision based on the properties of a sample. The implementation in this thesis works with density functions that are approximated on a grid. The input variables for these functions are the numeric values of the sample features. To approximate a function that depends on n input variables a n-dimensional grid is used. In spam recognition a sample could be a message. Its considered features could be the number of letters, the usage of punctuation, and the number of occurrences of certain buzzwords. In this case the function would have three input variables and the approximation grid would therefore be three-dimensional.
The number of points on a full grid with a given mesh wide grows exponentially with the number of dimensions. Solving a problem like function approximation on all grid points may be feasible for three-dimensions but certainly is not for much higher dimensions. This is called the curse of dimensionality and all grid based approaches to high dimensional problems suffer from it.
Sparse grids tackle the curse of dimensionality by omitting most of the full grids points. They try to consider only grid points that have a big impact on the overall solution. An efficient way to calculate the solution on a sparse grid is the sparse grid combination technique. It composes the sparse grid of small full grid components. This has many advantages compared to the direct calculation of the whole grid. Most of them result from the fact that the solution for the components can be computed independently of each other.

The accuracy of density function approximation and thus the accuracy of the classification is based on the mesh size of the sparse grid. A finer mesh can provide a better accuracy but also results in a higher number of grid points. For that reason, a typical learning algorithm starts with a coarse mesh and refines it as long as profitable. The problem here is that different features may contain different amounts of information. An algorithm may reach a point where a further refinement of a certain dimension would be a waste of grid points while refining another one would still provide accuracy improvements.
Refining only certain dimensions is called dimensional adaptation. The sparse grid combination technique is very suitable for that purpose. The independence of the component grids allows dimensional adaptive refinements by including additional components to the solution without needing to resolve the already existing ones.
The existing approach to density estimation based classification in the data mining pipeline of SG++ worked by directly computing the density function on the overall sparse grid. Over the course of this theses it was extended and now also supports the dimensional adaptive sparse grid combination technique.

2. Theoretical Background

This Chapter describes the technical background of the implementation. It starts by an introduction of the machine learning tasks density estimation and classification. Afterwards there is an explanation of how function approximation works on grids and how sparse grids are able to radically reduce the number of used grid points with out loosing a lot of information. Based on that the dimensional adaptive sparse grid combination technique is explained. The Chapter ends with a more detailed explanation about how density functions are approximated on a sparse grid.

2.1. Machine Learning

Machine learning is algorithmic knowledge generation from data. Learning algorithms train a model by generalizing information that is obtained from sample data sets. The goal is to make statements over new unseen data samples. Machine learning tasks are distinguished into supervised learning, unsupervised learning, and reinforcement learning. Supervised learning algorithms train on data sets that contain pairs of input data and the corresponding correct answer. In contrast to that unsupervised learning algorithms work only on input data and try to detect commonalities and differences between the data samples. Reinforcement learning works with software agents who are given positive or negative rewards, based on what actions they take in training situations [4]. The last type is not relevant for this thesis and just mentioned for the sake of completeness. Section 2.1.2 introduces the classic supervised learning task classification. The approach in this thesis is based on the unsupervised learning task density estimation, described in Section 2.1.1.

2.1.1. Density Estimation

Density Estimation is the task of approximating a density function. A density function of a continuous multidimensional random variable X describes the relative probability for the variable to have the vector of values $\vec{x}_i$ in the sample space it is evaluated on. The absolute probability at each point is of course zero because there is an infinite number of values a continuous variable can take. Density functions are often used to

obtain the probability for X to fall into a certain space of values.

$$P[X \in A] = \int_{x \in A} f(x)dx \tag{2.1}$$

When used for classification we evaluate the corresponding density functions of all the possible classes on the sample point and compare their results. The classification function can be formulated as finding the class which density function returns the highest probability for the given sample [4]. How density estimation functions are calculated on sparse grids will be described in Section 2.3, after the introduction of sparse grids.

2.1.2. Classification

Classification is the task of deciding to which class y_i a given sample $\vec{x}_i$ belongs. The sample $\vec{x}_i$ consists of a series of values, its features, and can be expressed as a vector. The space of classes is a finite set of names and can be mapped onto a subset of the natural numbers $K = \{1, ..., k\} \subset \mathbb{N}$. A classification miner tries to approximate a function f that assigns the correct class label to every combination of the d values in the sample vector.

$$f : \mathbb{R}^d \to K \tag{2.2}$$

As mentioned in Section 2.1 classification is a supervised learning problem. Thus, the training data set S consists of the data sample vectors and the corresponding classes they belong to.

$$S = \{(\vec{x}, y) | \vec{x} \in \mathbb{R}^d, y \in K, f(\vec{x}) = y\} \tag{2.3}$$

The SG++ toolbox works with a Bayes classifier based on class-conditional density functions $P[X = \vec{x} | f(\vec{x}) = y]$. The probability for a sample $\vec{x}_i$ to belong to a class y_i is obtained by applying the Bayes' theorem.

$$P[f(x_i) = y_i] = \frac{P[X = \vec{x}_i | f(\vec{x}_i) = y]}{P[f(X) = y_i]} \tag{2.4}$$

$P[f(X) = y_i]$ describes the probability for the occurrence of an element of the class y_i.

$$P[f(X) = y_i] = \frac{|\{(\vec{x}, y) \in S : y = y_i\}|}{|S|} \tag{2.5}$$

The classifier assigns a data sample $\vec{x}_i$ to the class $\vec{y}_i$ that maximizes $P[f(x_i) = y_i]$ [4].

2.2. Sparse Grid Based Function Approximation

Grid based function approximation is an easy straight forward way to approximate a function, for example the density function, arbitrary precisely by placing basis functions on grid points. In this chapter there will be a short introduction on how to approximate functions on a grid, the curse of dimensionality that occurs, and how it can be tackled using sparse grids. For more details on sparse grids please refer to [2].

2.2.1. Approximation on a Full Grid

When approximating a d-dimensional function $f : \Omega \to \mathbb{R}$ we first restrict Ω to the part that is of interest to us. In the following we define $\Omega = [0,1]^d \subset \mathbb{R}^d$. Additionally we consider f as a function that is zero on the boundary of Ω.
To construct an approximation u of f we discretize Ω into $(2^n)^d$ equidistant grid points x_i with n as the level of discretization. The distance between two adjacent points in each dimension is 2^{-n}. Centered on this grid points we place suitable basis functions φ_i. Thus, u can be constructed as a weighted sum of the basis functions with weights α_i.

$$f(x) \approx u(x) = \sum_i \alpha_i \varphi_i(x)$$

The basis function φ_i, which is centered at the grid point x_i and supports $1/\mu$ of Ω, can simply be derived by shifting and scaling the standard basis function that is placed in the center of the grid and supports the whole Ω.

$$\varphi_i(x) = \varphi(\mu x - x_i)$$

When implementing this, all that has to be saved are the basis function and a d-dimensional matrix that contains the weights α_i. When working with standard full grids, the number of points, and therefore the coefficients that have to be calculated and saved, grows exponentially with d and n. As mentioned in Chapter 1, this is known as the curse of dimensionality. It can be tackled using sparse grids instead of full grids.

2.2.2. Hierarchical Decomposition

Sparse grids are based on hierarchical decomposition of the approximation space Ω. In the case of function approximation that means there are different levels of functions that support different fractions of Ω. A basis function of level l supports $(-2)^l$ of Ω. The functions with level $l = k + 1$ refine the weighted sum of the functions of level $l \leq k$. Let us first explain this for one-dimensional functions and afterwards transfer it to the multidimensional case.

Hierarchical Decomposition in one Dimension

Let us choose the standard hat function Λ as an example basis function.

$$\Lambda(x) = max(1 - |x|, 0) \tag{2.6}$$

We can control the size of its support depending on its level l by dilatation and move its center by horizontal translation to the point $l * 2^{-i}$ that corresponds to the index i.

$$\Lambda_{l,i}(x) = \Lambda(2^l x - i) \tag{2.7}$$

To define which level contains which indexes we define the hierarchical index set I_l.

$$I_l = \{i \in \mathbb{N} : 1 \leq i \leq 2^l - 1, \, i \text{ odd}\} \tag{2.8}$$

The set of basis functions of level l can be used to describe a subspace $W_l \in \Omega$.

$$W_l = span\{\Lambda_{l,i} : i \in I_l\} \tag{2.9}$$

Figure 2.1 displays W_l for $1 \leq l \leq 3$ schematically. The space of functions V_n constructed of the hierarchical basis functions on a full grid for a defined level n follows to the direct sum of W_l.

$$V_n = \bigoplus_{l \leq n} W_l \tag{2.10}$$

Figure 2.2 shows a function $u \in V_3$.

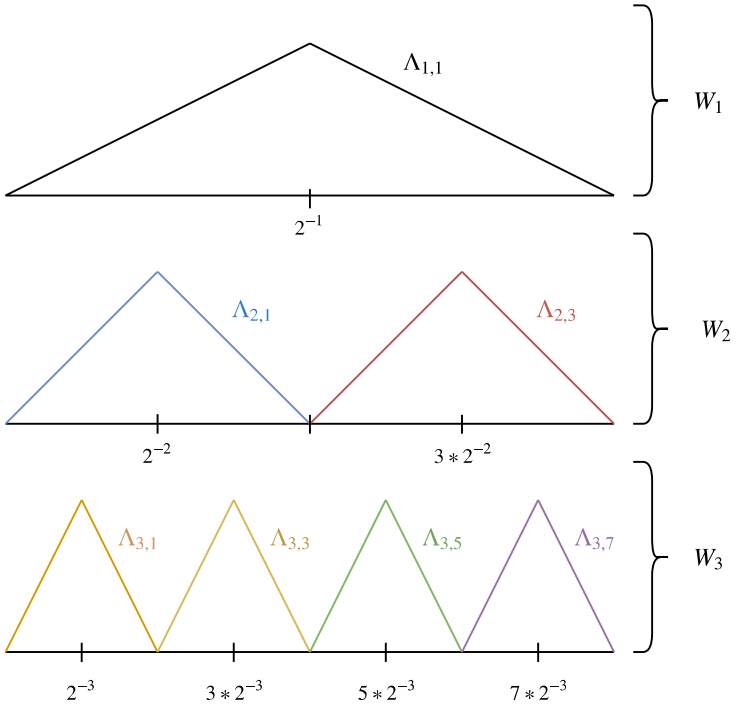


Figure 2.1.: Hierarchical one-dimensional hat basis function sets W_l for $l \leq 3$ on the approximation space $\Omega = [0, 1]$.

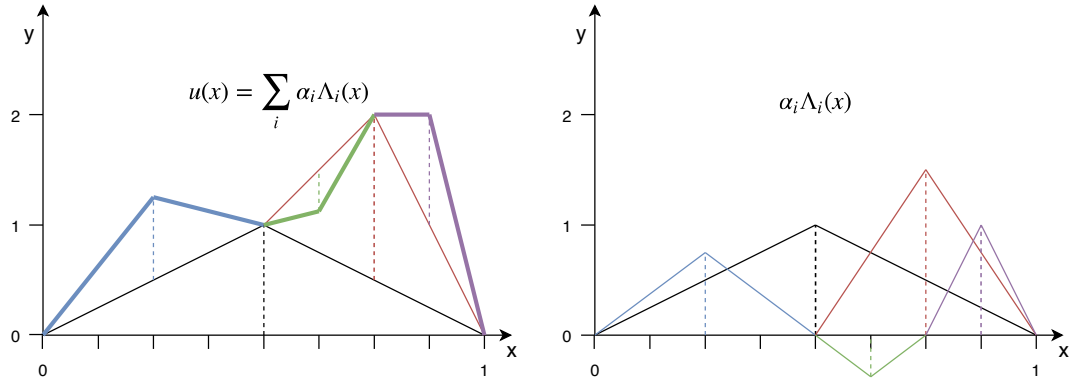


Figure 2.2.: Some function $u \in V_3$. The height of the hat functions Λ_i correspond to its weight α_i and is also called surplus.

Hierarchical Decomposition in more Dimensions

For the definitions in this chapter we need the l_∞-norm.

$$|\vec{l}|_\infty = \max_{1 \leq j \leq d} |l_j| \tag{2.11}$$

In the d-dimensional case the level l and the index i become level and index vector $\vec{l}$ and $\vec{i}$ that contain the levels and indexes for each dimension. The index set I_l follows to $I_{\vec{l}}$.

$$I_{\vec{l}} = \{\vec{i} \in \mathbb{N}^d : 1 \leq i_j \leq 2^{l_j},\ i_j \text{ odd}, 1 \leq j \leq d\} \tag{2.12}$$

The d-dimensional basis functions are constructed via a tensor product of the one dimensional basis functions.

$$\varphi_{\vec{l},\vec{i}}(\vec{x}) := \prod_{j=1}^{d} \varphi_{l_j,i_j}(x_j) \tag{2.13}$$

The subspace $W_{\vec{l}}$ follows accordingly.

$$W_{\vec{l}} := span\{\varphi_{\vec{l},\vec{i}}(\vec{x}) : \vec{i} \in I_{\vec{l}}\} \tag{2.14}$$

Figure 2.3 displays W_l for the case of the two dimensional hat function for $l_i \leq 3$ schematically. The space of functions V_n with maximal refinement level n is again written as a direct sum.

$$V_n = \bigoplus_{|\vec{l}|_\infty \leq n} W_{\vec{l}} \tag{2.15}$$

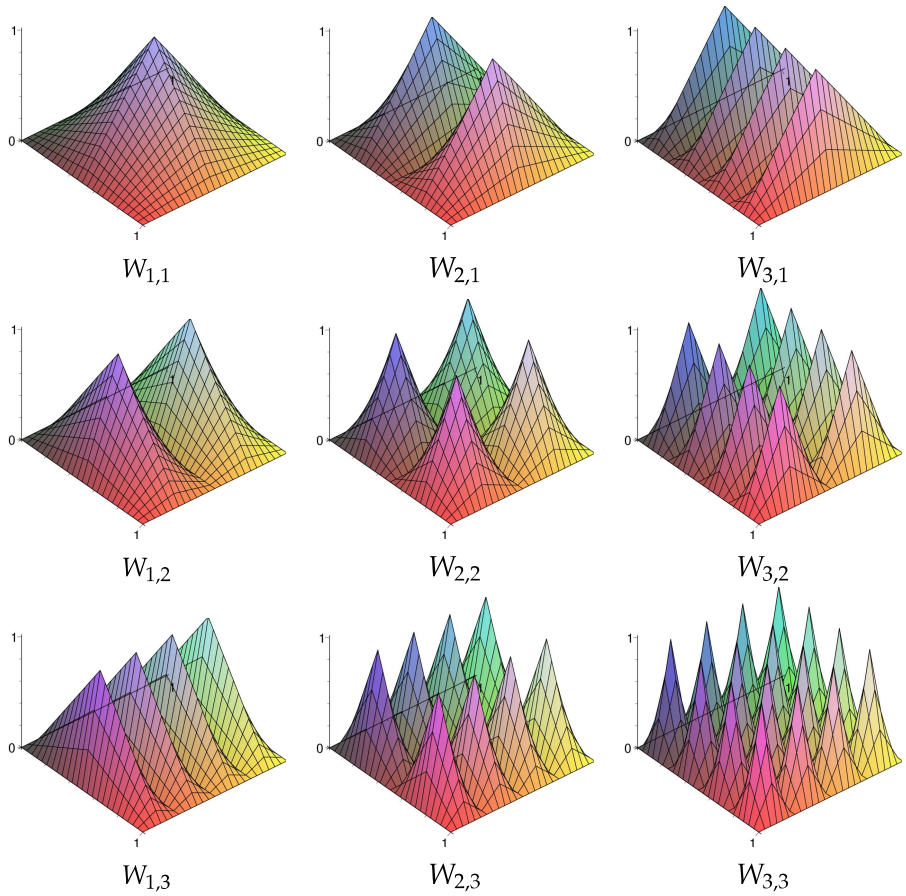


Figure 2.3.: Hierarchical two-dimensional hat basis function sets $W_{\vec{l}}$ for $l_j \leq 3$ [11].

2.2.3. Sparse Grids

For the definitions in this chapter we need the l_1-norm.

$$|\vec{l}|_1 = \sum_{j=1}^{d} |l_j| \tag{2.16}$$

The sparse grid space $V_n^{(1)}$ is obtained by selecting those subspaces $W_{\vec{l}}$ that contribute most to the total solution while containing as little basis functions as possible.

$$V_n^{(1)} := \bigoplus_{|\vec{l}|_1 \leq n+d-1} W_{\vec{l}} \tag{2.17}$$

Figure 2.4 shows the construction of $V_3^{(1)}$ in the two-dimensional case. Note that in the one dimensional case there is no difference between the full and the sparse grid. For two dimensions the difference lays in the gray subspaces. Generally speaking, the number of grid points for a given discretization level n is reduced from $\mathcal{O}((2^n)^d)$ to $\mathcal{O}(2^n n^{d-1})$, while the asymptotic accuracy only falls slightly from $\mathcal{O}(2^{-2n})$ to $\mathcal{O}(2^{-2n} n^{d-1})$ [11]. Figure 2.5 shows a three-dimensional sparse grid of level 6.

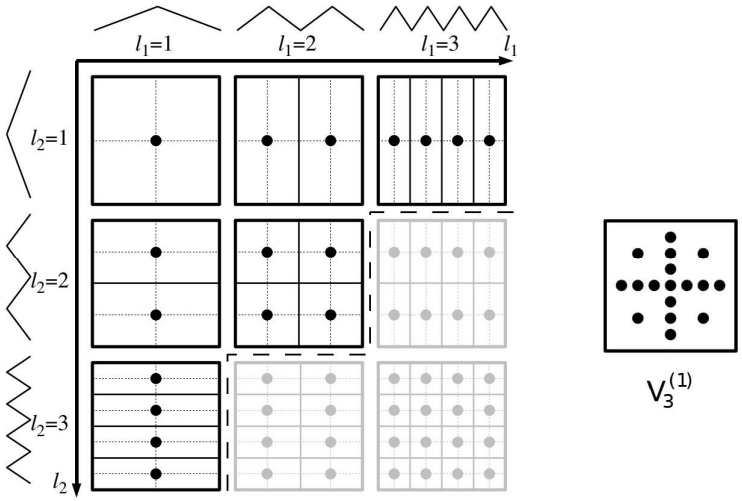


Figure 2.4.: Two-dimensional sparse grid space $V_3^{(1)}$ and the corresponding subspaces $W_{\vec{l}}$ it consists of. Adding the gray subspaces would form the full grid [11].

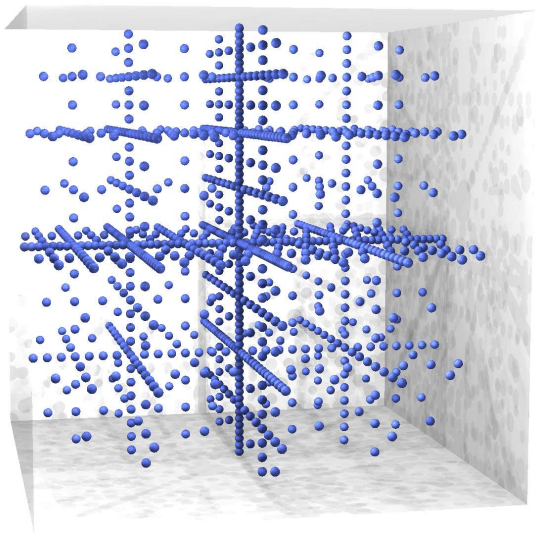


Figure 2.5.: Three-dimensional sparse grid of level 6 [11].

2.2.4. The Sparse Grid Combination Technique

A disadvantage of sparse grids compared to full grids is that they need more complicated algorithms to solve problems on them. In our case solving means calculating the surpluses of the basis functions centered on the grid points. The sparse grid combination technique circumvents this by composing the sparse grid of small anisotropic full grid components. An anisotropic full grid is a full grid whose level of refinement varies between the dimensions and can be expressed as $u_{\vec{l}} \in V_{\vec{l}}$. To evaluate the sparse grid function the component grid functions are evaluated individually, weighted, and summed up. The solutions of the component grids are calculated independently of each other. This allows to use algorithms designed to work with full grids and to parallelize their computation. Also, as explained later in Section 2.3, approximating on a grid has a complexity of $\Omega(n^3)$ with n as the number of grid points. Therefore its cheaper to calculate the solution on a series of small grids than on one big grid. Thus, using the combination technique can further reduce the complexity of the overall solving algorithm.

The Standard Combination Technique

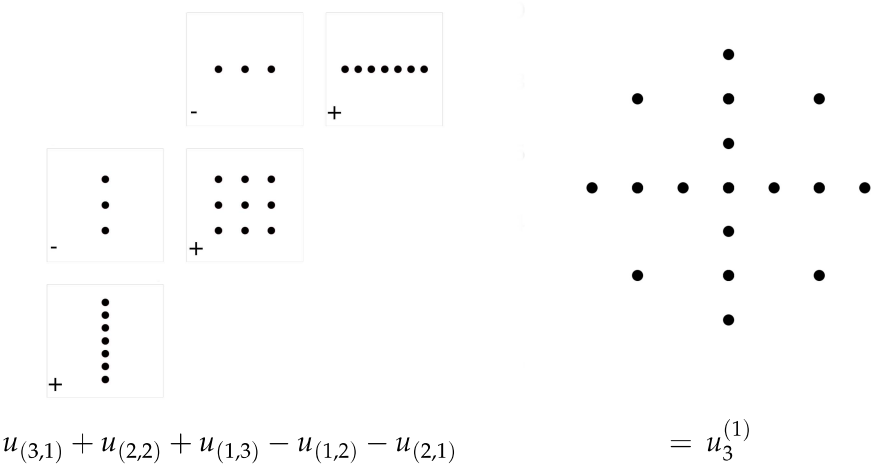
The standard combination technique calculates a solution $u_n^{(1)}$ from the d-dimensional sparse grid space $V_n^{(1)}$.

$$u_n^{(1)} = \sum_{q=0}^{d-1} (-1)^q \binom{d-1}{q} \sum_{\vec{l} \in I_{n,q}} u_{\vec{l}} \tag{2.18}$$

$I_{n,q}$ is a fixed set of level vectors that defines the components that are used.

$$I_{n,q} = \{l \in \mathbb{N}_0^d : |\vec{l}|_1 = n + d - 1 - q\} \tag{2.19}$$

Figure 2.6 shows the necessary components to form a regular, that means uniformly refined in all of its dimensions, two-dimensional sparse grid of level three. For two dimensions the used components in the standard combination technique lie on two diagonals, for three dimensions they lie on a three plane and for higher dimensions they lie on multiple hyper planes. The standard combination technique therefor always results in a regularly formed sparse grid structure [8].



$$u_{(3,1)} + u_{(2,2)} + u_{(1,3)} - u_{(1,2)} - u_{(2,1)} = u_3^{(1)}$$

Figure 2.6.: The sparse grid combination technique for a $u_3^{(1)} \in V_3^{(1)}$ for two dimensions. The components are added up accordingly to their sign. In higher dimensions the coefficients also take different values then -1 and 1.

Dimensional Adaptive Combination Technique

A sparse grid that provides the same level of discretization in every dimension is not always the optimal solution. Remember that in data driven learning applications the

different dimensions stand for different features. Some of them may be more important than others. All of the information that can be derived from some feature might already be accurately enough expressed by some discretization level n, while for an other feature a finer grading would still deliver additional information. To tackle this problem, the sparse grid combination technique can be generalized to not only allow using component grids from under some hyper planes, like in Equation 2.19, but to work with the following admissibility condition [5].

$$(\vec{k} \in I \wedge \vec{j} \leq \vec{k}) \rightarrow \vec{j} \in I \tag{2.20}$$

Note that $\vec{a} \leq \vec{b}$ connotes component-wise comparison. Now arbitrary components can be added to refine certain dimensions as long as the admissibility condition holds. To compute the corresponding coefficients $c_{\vec{l}}$ following equation can be used [8].

$$c_{\vec{l}} = \sum_{\vec{l} \leq \vec{i} \leq \vec{l}+\vec{1}, i \in I} (-1)^{|\vec{i}-\vec{l}|_1} \tag{2.21}$$

Figure 2.7 shows three admissible component sets and their coefficients for two-dimensional sparse grids. The left one equals the standard combination technique and results in a regular sparse grid. The other two result in also valid but anisotropic sparse grids.

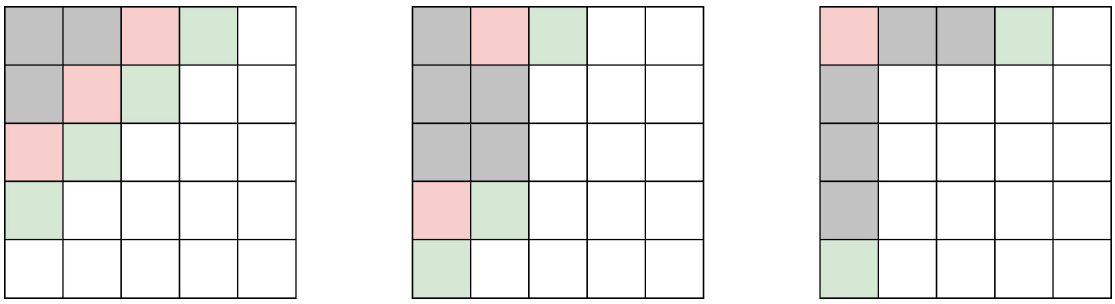


Figure 2.7.: Three admissible sets of component grids. The left set forms a regular sparse grid, while the other two form anisotropic sparse grids. Green: $c_{\vec{l}} = 1$, Red: $c_{\vec{l}} = -1$, Gray: $c_{\vec{l}} = 0$.

A dimensional adaptive algorithm step by step adds new component grids without violating the admissibility condition until some global criterion is fulfilled. For choosing which component to add next, a greedy approach is used. Thus, we need to find a way to estimate the error a component grid has, while considering the costs of refining it. One approach could be to assume that the component that had the greatest impact when added still needs the most additional refinement. On example of estimating a

components impact is to take the l_2-Norm of its surpluses, mentioned in Section 2.2.2. As an estimation for its refinement costs we could use its number of basis functions, i.e. the component grids number of points.

2.3. Density Estimation on Sparse Grids

Approximating a density function on a sparse grid equates to an optimization problem with two criteria. Want the to extract as much information from the training data as possible while staying general enough to work with new unseen data. The first criterion can be tackled by minimizing the squared difference to a guessed function f_e, which is in most cases highly overfitted. This can for example be a kernel density function f_K. A kernel density function is constructed by placing kernel functions K on the data points x_i from the training data set S and evaluating them all together [10].

$$f_K(x_i) = \frac{1}{|S|h} \sum_{x_i \in S} K(\frac{x - x_i}{h}) \tag{2.22}$$

The smoothing coefficient h is used for scaling the kernel functions. There are infinite possible kernel functions, but the most commonly used is the Gaussian kernel.

$$K(x) = (2\pi)^{-1/2} \mathrm{e}^{-x^2/2} \tag{2.23}$$

We can also work with Dirac delta functions δ_{x_i} centered at the training data points $x_i \in S$. Note that this is somehow similar to setting the smoothing coefficient h of the Gaussian kernel very close to zero.

$$f_\delta(x) = \frac{1}{|S|} \sum_{x_i \in S} \delta_{x_i}(x) \tag{2.24}$$

Staying general in the density estimation case means to allow some derivation from the training data samples. Note that a perfect adaptation to the training data would result in a density function that evaluates to zero for every data point but the ones from the training data. This would be useless for a continuous observation space because the chances of hitting the same point in a continuous space is zero. Because we are talking about functions, we can define generality as smoothness. Fulfilling the generalization criterion therefore can be realized by minimizing the squared derivation. The searched density function $p(x)$ is then obtained by solving the following optimization problem in the sparse grid function space $V^{(1)}$.

$$p(x) = \underset{u^{(1)} \in V^{(1)}}{\arg\min} \int_\Omega (f(x) - f_e(x))^2 \mathrm{d}x + \lambda \int_\Omega f''(x)^2 \mathrm{d}x \tag{2.25}$$

The coefficient λ controls the proportion of smoothness and accuracy. The equation can be transformed to the following variational equation with $W^{(1)}$ as the space of hierarchical basis functions of $V^{(1)}$ with the corresponding index set $I^{(1)}$[6].

$$\int_\Omega p(x)(\varphi(x) - f_\epsilon(x))\mathrm{d}x + \lambda \int_\Omega p''(x)\varphi''(x) = 0, \qquad \forall \varphi \in W^{(1)} \tag{2.26}$$

Working with Equation 2.24 it can farther be transformed to

$$\int_\Omega p(x) = \sum_{i \in I^{(1)}} \alpha_i \varphi_i \varphi(x)\mathrm{d}x + \lambda \int_\Omega p''(x)\varphi''(x) = \frac{1}{|S|} \sum_{x_i \in S} \varphi(x_i), \qquad \forall \varphi \in W^{(1)} \tag{2.27}$$

and be expressed as a system of linear equations

$$(\boldsymbol{R} + \lambda \boldsymbol{C})\vec{\alpha} = \vec{b} \tag{2.28}$$

where $R_{ij} = \langle \varphi_i \varphi_j \rangle_{L2}$, $C_{ij} = \langle \varphi_i'' \varphi_j'' \rangle_{L2}$ and $b_i = \frac{1}{|S|} \sum_{x_i \in S} \varphi_i(x_j)$. Solving this system we obtain the vector $\vec{\alpha}$ containing the coefficients of the corresponding basis functions that form the sparse grid density function. The matrices **R** and **C** are of size $n \times n$ with n as the number of points in the sparse grid. Therefore the complexity of solving the equation system is in $\Omega(n^3)$ and only depends on the dimension and discretization level of the sparse grid, not on the size of training data.

3. Implementation

This Chapter describes the implementation of a combination grid based density estimation model fitter that supports dimensional refinements and its integration to the data mining pipeline of SG++.

3.1. Introduction of SpACE: Spatial Adaptive Combination Environment

The Spatially Adaptive Combination Environment is a framework, created by Michael Obersteiner, that implements different variants of the spatial adaptive combination technique. While it was originally designed for numerical integration it currently is getting extended to allow a more general usage of the sparse grid functionality. The dimensional adaptive refinement used in this thesis was implemented by Hendrick Möller over the course of his bachelor thesis [7]. For our implementation we just use the class CombiScheme, which is responsible for the calculation of the level vector sets and their coefficients as described in Section 2.2.4.

3.2. Introduction of SG++: General Sparse Grid Toolbox

The General Sparse Grid Toolbox is an open source framework for sparse grid based applications. It is written in C++ and was created by Dirk Pflüger as part of his dissertation [11]. Currently the chair of Scientific Computing at the Technical University of Munich and the Institute for Parallel and Distributed Systems at the University of Stuttgart are responsible for its further development. One of its modules is the data mining pipeline that was designed for data-driven tasks. It contains an approach to classification based on sparse grid density estimation as introduced in Section 2.1.2. It was implemented by Dominik Fuchsgruber and is described in his bachelor thesis [4]. This implementation was extended by the dimensional adaptive sparse grid combination technique using some of the functionality of the Spatial Adaptive Combination Environment that was introduced in Section 3.1.

3.3. The SG++ Data Mining Pipeline

For users of the data mining pipeline all of its functionality is encapsulated in sparse grid miners as displayed in Figure 3.1. A miner consists of a data source, a model fitter, and a scorer. The data source manages the splitting of the data set into training and validation data, and into batches of arbitrary size for the learning process. It also sorts the data samples according to their class labels and provides functionality to shuffle them. The model fitter handles the training of the model using the training data. The scorer evaluates the model on the validation data and on the training data itself and delivers information about the quality of the fit. Testing on the training data may seem superfluous, but is reasonable considering that the fitter does not just imitate the training data but also generalizes it. There are two different kinds of sparse grid miners that only differ in their kind data source. They are called SparseGridMinerSplitting and SparseGridMinerCrossValidation. The first just splits the data set into two complementary parts. One is used for training and one exclusively for validation. The second miner type uses cross validation. The basic idea here is that, especially in small data sets, the validation data may differ strongly from the training data. Therefore the process of training and validation is performed multiple times for different splits of the data set.

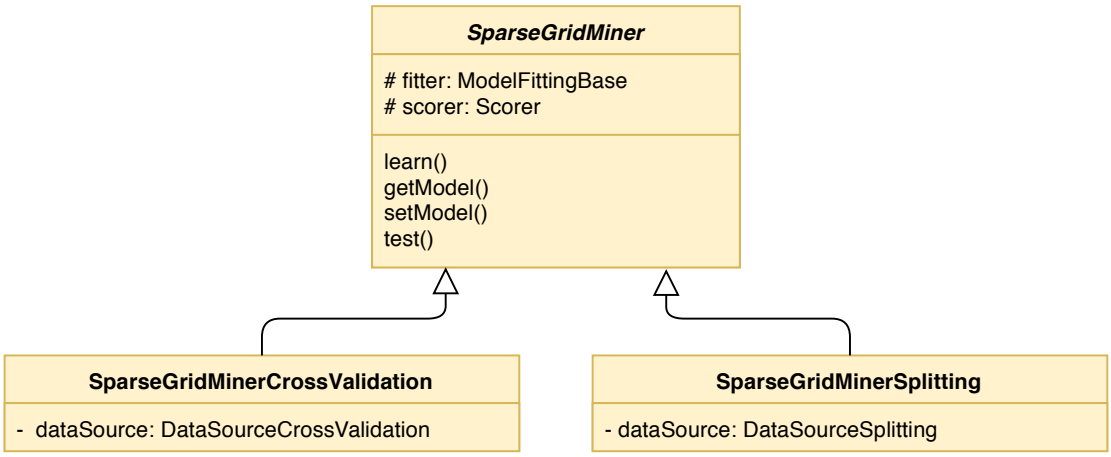


Figure 3.1.: Simplified class diagram of the two kinds of sparse grid miners, both inheriting from the abstract *SparseGridMiner*.

The miners are built in miner factories. The type of miner factory that is used determines the type of machine learning task that is performed. So far there is classification, density estimation, and least square regression. All further configuration is done via a JSON configuration file that is getting parsed during the creation of the miner. The structure

of these settings is described in Section 3.3.1.

3.3.1. The Configuration Structure

The configuration structure follows the structure of the sparse grid miner and is therefore divided into the configuration of the data source, the scorer, and the fitter. There exist some more configuration structures, which we will ignore in this theses because they are not relevant for us. The data source configuration contains the path to the data set and information about how to interpret, divide, and shuffle it. The scorer configuration contains the metric that is used to calculate the quality of the fit. The fitter configuration is divided into a series of sub structures and contains, among others, the properties of the initial grids, the adaptive behavior, the density estimation technique, and the kind and strength of the applied regularization. So far the grid configuration structure only allowed to store a level of discretization that is equal in all dimensions. This was expanded to additionally store a level vector for the anisotropic case.

3.3.2. Structure of the Model Fitters

A simplified class diagram of the classification and density estimation model fitters is displayed in Figure 3.2. A model fitter consists of its configuration, see Section 3.3.1, its data set, and a series of functions controlled by the miner to train the model:

- *fit()*: Fits the initial model to the training data.
- *refine()*: Improves the model by adaptive refinement of the underlying grids.
- *update()*: Trains the existing model with the given data samples.
- *evaluate()*: Evaluates the model on one or more data points.
- *reset()*: Resets the model to its initial state.

A model fitter for density estimation operates on a single grid that represents the estimated density function. It therefore inherits from an abstract class that encapsulates general functionality for single grid model fitters and contains the function *getSurpluses()*. It returns the surpluses that correspond to the α_i in Section 2.2 and will be used for refinement decisions in the dimensional adaptive combination technique. The two types of density estimation miners provided so far only differ in their way of solving the linear system for calculating the weights of the basis functions that approximate the density function, see Section 2.3.
A classification model fitter does not operate on a single grid but consists of a vector of density estimation model fitters each corresponding to one class label. It also

implements the inherited abstract functions of the model fitting base by training and evaluating its density estimation model fitters, compare to Section 2.1.2.
The newly added model fitter is called ModelFittingDensityEstimationCombi and uses the sparse grid combination technique. It holds a vector of density estimation model fitters, but the models here do not contain density functions for different classes. They are the component grids for the combination technique and are fitted independently but evaluated together. To evaluate the overall density function at a certain point, all the component functions are evaluated, weighted with the corresponding coefficients, and summarized as explained in Section 2.2.4. The new fitter inherits from the abstract density estimation fitter class and can therefore be used by the classification fitter without any additional changes.

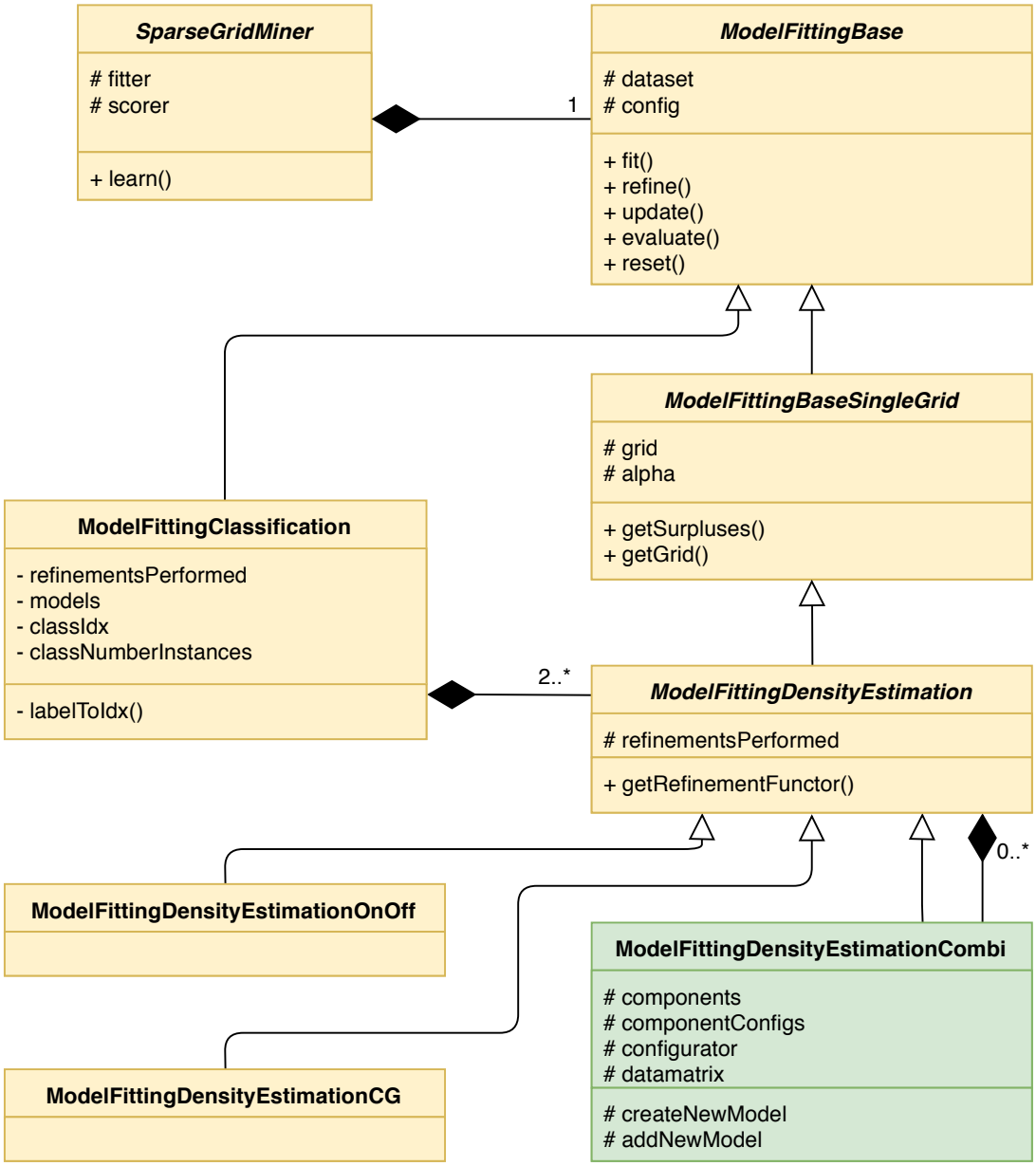


Figure 3.2.: Simplified UML class diagram of the different kinds of model fitters and their relation. The class ModelFittingDensityEstimationCombi was added.

3.3.3. The Dimensional Adaptive Combination Technique

As mentioned the new model fitter holds a vector of density estimation model fitters called components. Each component holds an anisotropic full grid that is fitted individually to the training data. The level vectors to define the discretization levels of components and their coefficients for the combined evaluation are obtained from a class in SpACE, called CombiScheme.
Because SpACE is written in Python and SG++ is written in C++ an adapter is needed to communicates between them. Therefore the class CombiConfigurator was created. It uses the Python/C API to store an instance of CombiScheme as a member variable and call its methods. It also converts data types used in SG++ to the ones used in SpACE and vice versa.
Figure 3.3 shows a simplified sequence diagram of the interactions between the model fitter, the adapter class, and CombiScheme during a learning process. First of all an instance of CombiScheme is created, the configuration for the initial component set obtained, and the components created. Now the refinement process starts and is reiterated as often as defined in the adaptivity configuration, see Section 3.3.1. As explained in Section 2.2.4 the refinable component with the greatest error is chosen to be refined. Note that refining a component does not change its grid but results in changes of the component set and its coefficients. Depending on the situation additional components get added, existing ones removed, or their combination coefficients changed. The error is currently defined as the L2-Norm of the surpluses of a component divided through its number of grid points. This is a reasonable measurement considering the approximation of a density function, but certainly not the ideal choice for every use case. In future implementations factors different from producing the best possible density function approximation should be considered. For example in the case of classification the refinement algorithm should focus on the areas in which the different classes are hard to distinguish.

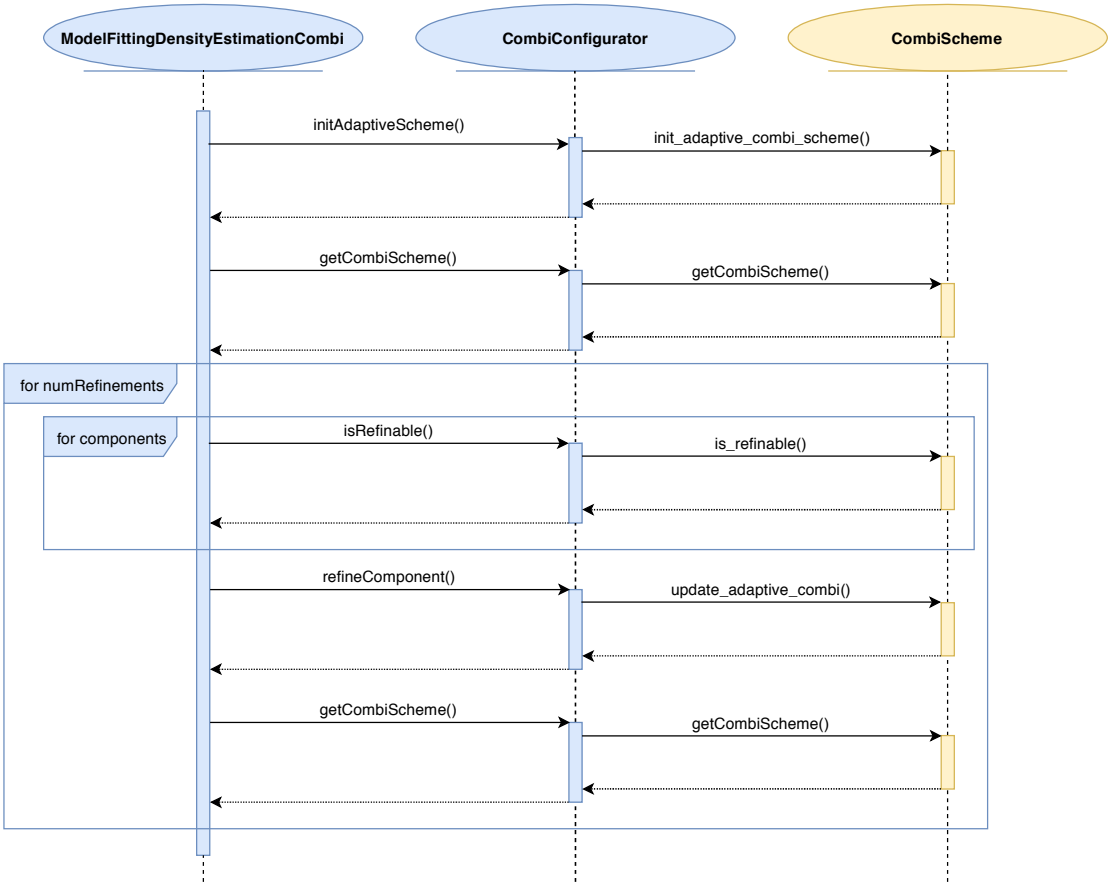


Figure 3.3.: Simplified sequence diagram of an instance of *ModelFittingDensityEstimationCombi* during a learning process.

3.3.4. Anisotropic Full Grids

Fundamental for the sparse grid combination technique are anisotropic full grids. Such were not supported so far and had to be added as an operation for the hash generator in the base module of SG++. It works by filling a hash storage with points which are defined by their dimension, level, and index.

4. Evaluation

In this Chapter the correctness of the implementation is evaluated on four different data sets. Two of them were generated artificially, the other two contain real data. The new implementation is compared with the old one in terms of speed and classification accuracy.
From the two model fitters for the density estimation the, according to the evaluation in [4], slightly better Online/Offline splitting based approach was used for all tests. The data sets were each separated into two halves one used for training and one used for validation. The separation was performed randomly for each test run to guaranty trustworthy results. The algorithms always started from a level one grid and used the whole training data at each step. Batch learning does not make sense for the dimensional adaptive combination technique because it would lead to component grids that where fitted to different data but are evaluated together. The configuration files that contain for example the regularization strength, the refinement strategy for the old implementation, and the kind of matrix decomposition used can be found in the appendix. The hardware specifications of the laptop that run the test as well as their results can also be found there. The variances were neglected in the plots to keep them more clear.
Note that the number of overall grid points that appears in this Chapter counts all grid points used for the calculation. Thus, in case of the sparse grid combination technique this number equals the sum of the components points and not the number of points the resulting sparse grid would have. Also this number does not denote the grid points used per class, but of all class conditional density functions summarized.

4.1. The Circles Data Set

The first data set was generated using scikit-learn [9]. It contains 500 two-dimensional data points distributed equally over two classes. As pictured in Figure 4.1 the classes form two noisy drawn circles that slightly overlap. The round shape may lead to the conclusion that the data set can be separated by a single grid point but that is not the case. A single grid point could only represent a density function with a high number of samples in the middle of the approximation space that decreases to the sides. Thus, while it is actually possible to define the two classes just by their range of distance from

the middle, it takes the implemented approaches a lot of grid points to achieve a good accuracy.

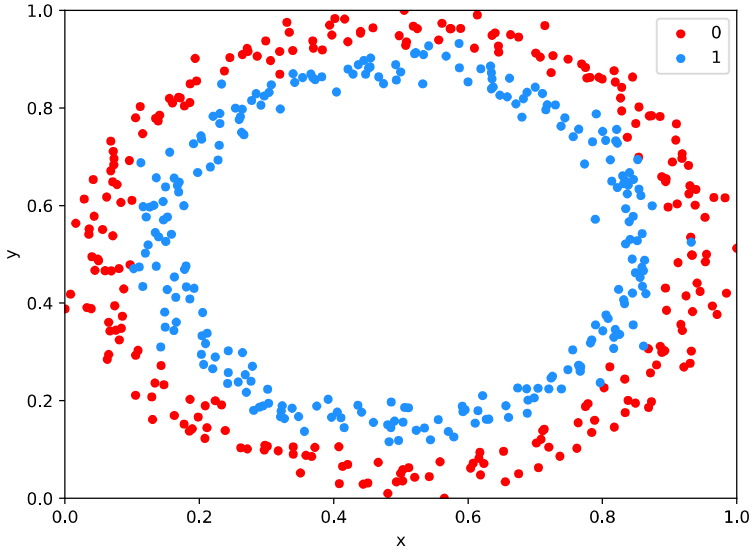


Figure 4.1.: Visualization of the circles data set. It contains 500 data points and was generated using scikit-learn [9].

The test results on this data set where averaged over 100 runs for each algorithm. The regularization strength was set to 0.01. Figure 4.2 shows the accuracy of the old and the new implementation in respect to the overall number of grid points. In the area from about 100 to 1000 grid points the old implementation outperforms the new one. This is due to the fact that the old implementation can adapt spatially to the circular shape. The new implementation can just refine whole dimensions. In this case both dimensions contain the same amount of information, so the refinement steps result in a regular sparse grid. Eventually the new implementation reaches a mesh size fine enough to a accurately classify. Starting at around 2000 grid points it even outperforms the old implementation. Note that the big difference between the accuracy on training data and on the validation data results from the small number of overall samples.
Figure 4.3 shows the time the algorithm needs to refit the grid to the training data after a refinement step. The new implementation clearly outperforms the old one here. Thanks to independence of the component grids, only newly added components has to be fitted after a refinement step. In contrast to this, adding points in the old implementation requires refitting the whole sparse grid. The impact of this difference grows with the overall number of grid points.

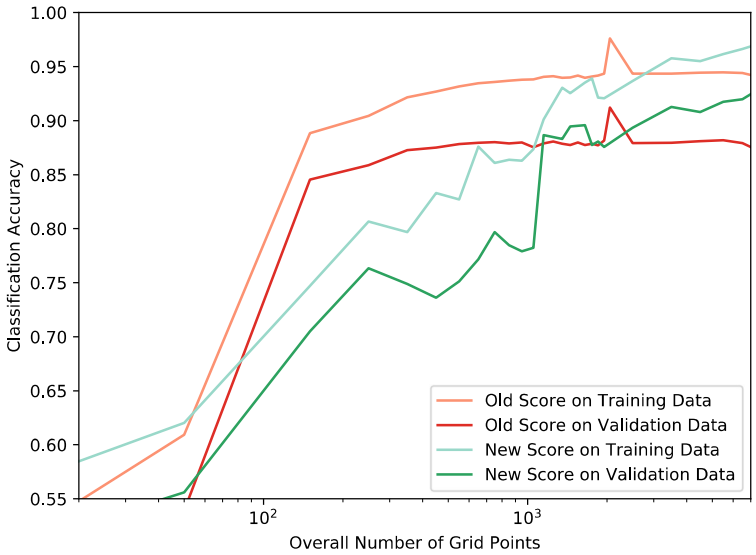


Figure 4.2.: Evaluation of the accuracy on the circles data set in respect to the number of grid points, averaged over 100 runs. Details in Appendix Section A.2.1.

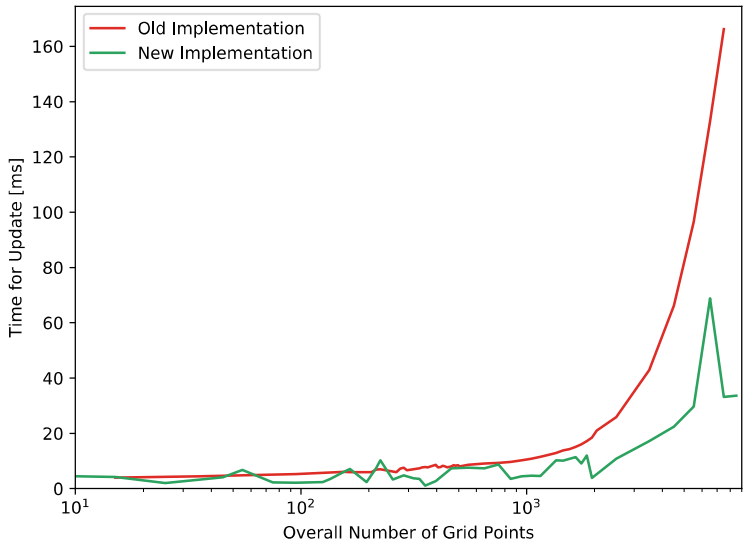


Figure 4.3.: Evaluation of the required time fit to the circles data set after a refinement of the model structure in respect to the number of employed grid points, averaged over 100 runs. Details in Appendix Section A.2.1.

4.2. The Funny Chess Data Set

This data set was also generated using scikit-learn [9]. It contains 5000 two-dimensional data points distributed unequally over three classes. The goal was to create a data set that requires a lot more refinements in one dimension than in the other. To separate the classes in terms of its x values a grid with mesh size 2^{-2} is sufficient. In the y direction there are 17 layers of classes why a much finer mesh size is necessary. A visualization of the data set is shown in Figure 4.4.

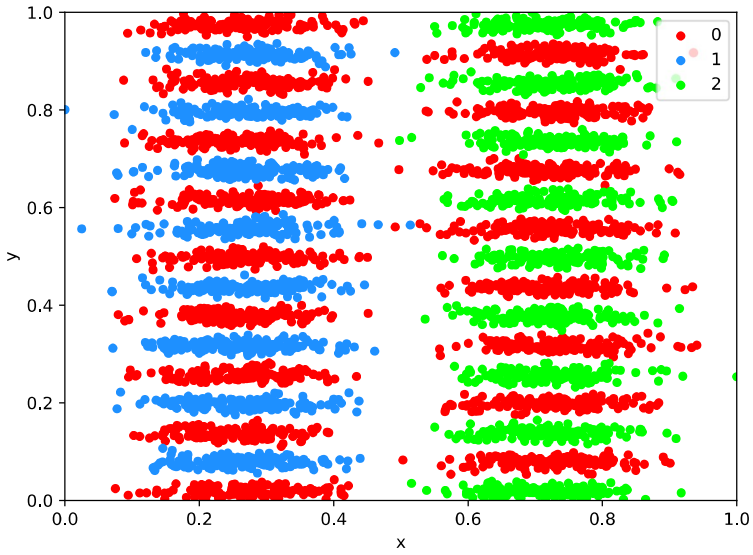


Figure 4.4.: Visualization of the Funny Chess data set. It was created using scikit-learn [9]. Class 0 contains 2500 data points, classes 1 and 2 each 1250.

The accuracy of the implementations on this data set is shown in Figure 4.5. The regularization strength was again set to 0.01 and the results again averaged over 100 runs. The dimensional adaptive refinements of the new implementation behave and perform just as expected. Up until around 600 grid points the new algorithm has no chance to approximate the class conditional density functions accurately and therefor the classification fails completely. The old implementation on the other hand is free to directly spend grid points where they are needed and thus its accuracy starts to improve earlier and more continuous. Once the necessary level of refinement is reached the new implementation overtakes the old one and reaches almost 100% accuracy. This is not surprising because the classes are almost fully separable and the data set is very suitable for dimensional adaptive refinements. Note the that small difference between

the score on the training data and on the validation data results from the big number of points and the little noise between the classes. It is not quite clear why the old implementation does not also come closer to an accuracy of 100%. The reason cannot just be over fitting, because then at least the accuracy on the training data would reach a higher value.
Figure 4.6 shows the necessary time for the update steps. There is no big difference to the circles data set here.

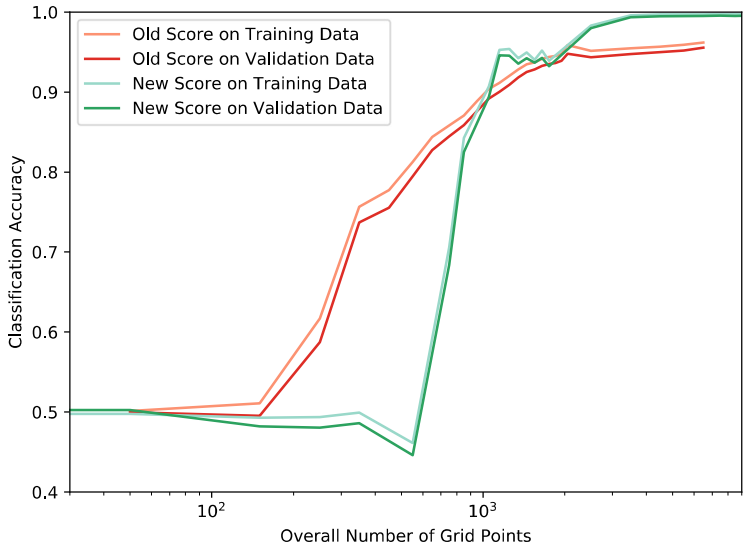


Figure 4.5.: Evaluation of the accuracy on the Funny Chess data set in respect to the number of grid points averaged over 100 runs. Details in Appendix Section A.2.2.

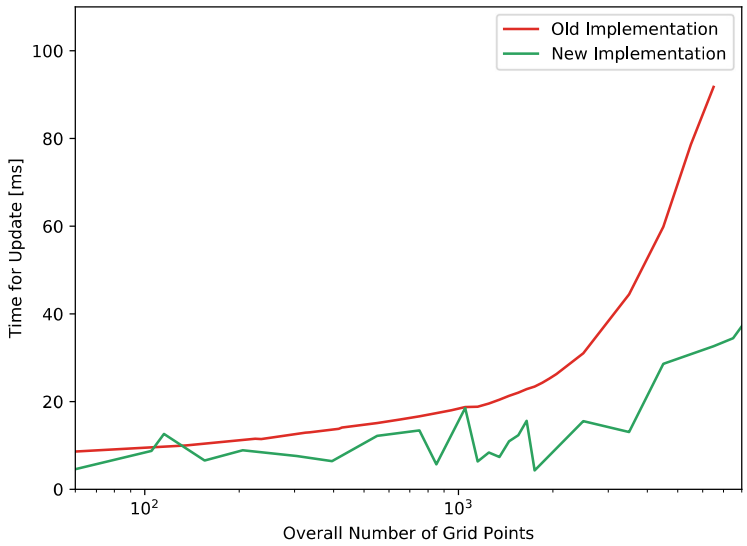


Figure 4.6.: Evaluation of the required fitting time after refinements of the model structure on the Funny Chess data in respect to the number of employed grid points averaged over 100 runs. Details in Appendix Section A.2.2.

4.3. Iris Flower Data Set

The Iris data set is the first real data set the implementation was evaluated on. It was introduced in 1936 by Ronald Fisher [3] and has become a typical test data set for classification. It contains 150 samples equally distributed over three different species of Iris, named Iris setosa, Iris virginica and Iris versicolor. It is a four-dimensional data set and the features are the sepals and the petals wide and length in centimeters. Because the implementations can only handle values between zero and one it had to be normalized.
Both implementations were able to reach a very high accuracy with a very small number of grid points as displayed in Figure 4.7. As before the regularization strength was set to 0.01 and the results averaged over 100 runs for each algorithm.
Figure 4.8 shows the averaged update time for this data set. Interesting to see here is that the increased number of dimensions result in a even earlier difference in the fitting time after refinements. Note that this must not be overrated because it results partly from the fact that the grid points are distributed over a larger number of components, from which only a small subset changes at each step.

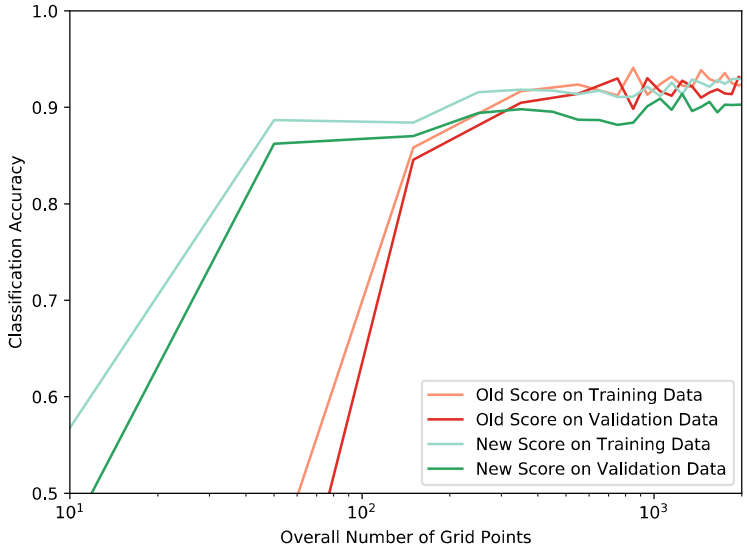


Figure 4.7.: Evaluation of the accuracy on the Iris data set in respect to the number of grid points averaged over 100 runs. Details in Appendix Section A.2.3.

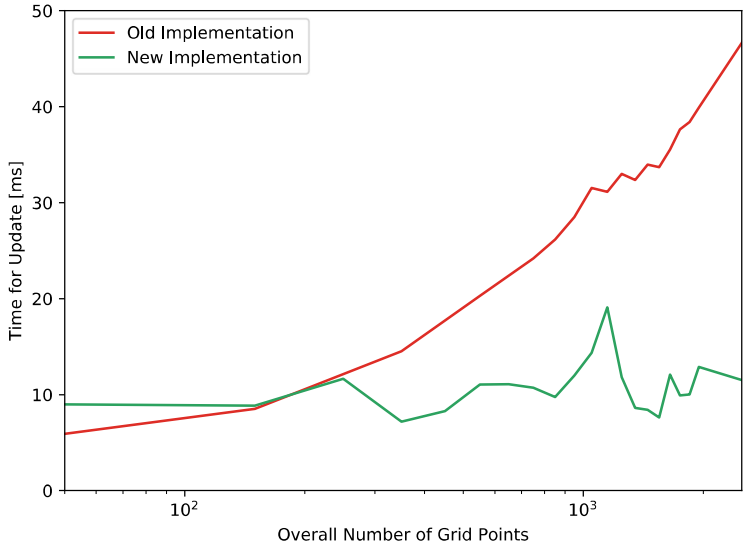


Figure 4.8.: Evaluation of the required fitting time after refinements of the model structure on the Iris data in respect to the number of employed grid points averaged over 100 runs. Details in Appendix Section A.2.3.

4.4. The DR10 Data Set

The last data set that the implementation was evaluated on is a part of the tenth data release of the Sloan Digital Sky Survey, DR10 [1]. The used part contains four-dimensional photometric measurements of 32317 stars and 32316 quasars.
The accuracy of the implementations on this data set is shown in 4.9. This time the regularization strength was set to 10^{-7}. Because the tests runs on this data set took much longer than on the other data sets the results were only averaged over 5 runs. The performance of both implementations was very unsteady, even after the averaging. It is not clear why, but the oscillations appeared in every single run. The very small difference between the score on the training and on the validation data indicates that it neither results from over fitting nor from a two small numbers of samples. It is possible that it just takes a much higher number of grid points to approximate the density functions accurately enough.
Figure 4.10 shows the time necessary for the refitting after a refinement step. In this data set the difference is clearer than in any of the others. The higher the number of overall grid points and dimensions results in a grid point distribution over a lot more component grids and therefor in a much smaller fraction that has to be refitted after each step.

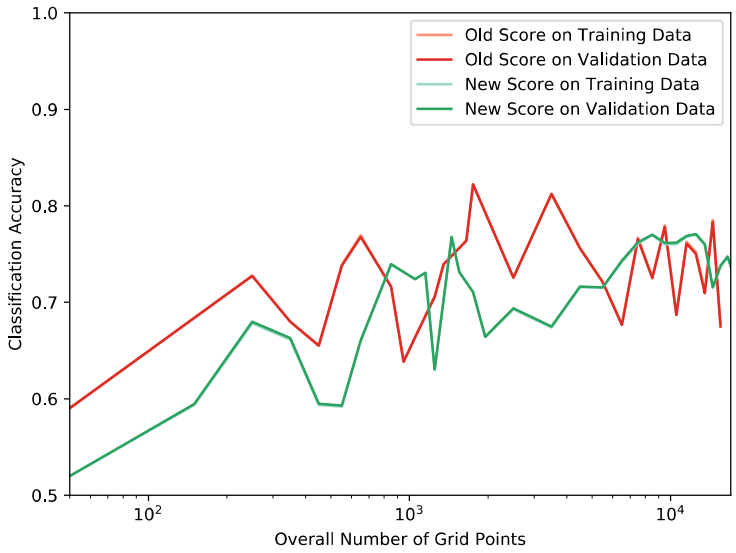


Figure 4.9.: Evaluation of the accuracy on the DR10 data set in respect to the number of grid points averaged over 5 runs. Details in Appendix Section A.2.4.

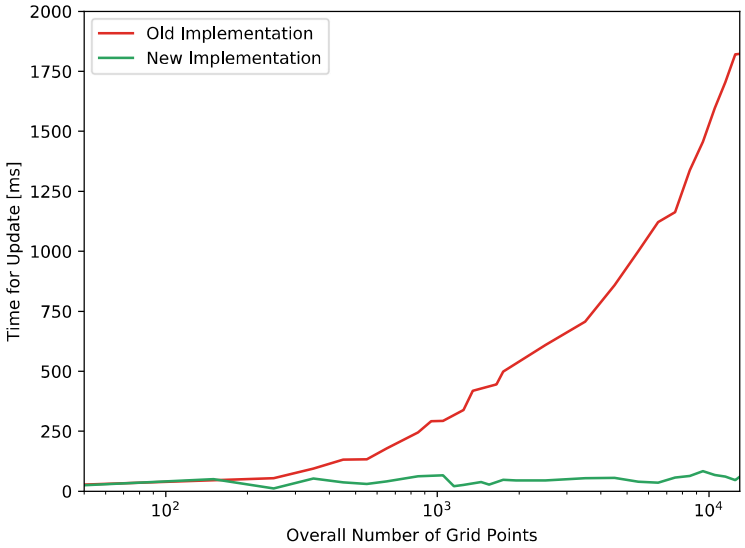


Figure 4.10.: Evaluation of the required fitting time after refinements of the model structure on the DR10 data in respect to the number of employed grid points averaged over 5 runs. Details in Appendix Section A.2.4.

5. Conclusion and Future Work

The dimensional adaptive combination technique was successfully added to the data-mining pipeline of SG++ in form of a new density estimation model fitter. Its main advantage is the independence of the component grids. It was used to provide economic dimensional refinements of the sparse grid structure by keeping the solution of already solved components, instead of recalculating the whole solution after every refinement step.
Still there are a lot of things future implementations could improve. As mentioned in Section 3.3.3 refinement decisions should not only be made with the goal to produce accurate density functions but particularly to separate classes from each other. Future implementations could also parallelize the computation of the component grid solutions. The sparse grid combination technique is not limited to dimensional adaptive refinements. Future implementations could extended the new model fitter to use spatial adaptive refinements as described in [8].

List of Figures

Bibliography

[1] C. P. Ahn, R. Alexandroff, C. A. Prieto, F. Anders, S. F. Anderson, T. Anderton, B. H. Andrews, É. Aubourg, S. Bailey, F. A. Bastien, et al. "The tenth data release of the Sloan Digital Sky Survey: first spectroscopic data from the SDSS-III Apache Point Observatory galactic evolution experiment." In: *The Astrophysical Journal Supplement Series* 211.2 (2014), p. 17.

[2] H.-J. Bungartz and M. Griebel. "Sparse grids." In: *Acta numerica* 13 (2004), pp. 147–269.

[3] R. A. Fisher. "The use of multiple measurements in taxonomic problems." In: *Annals of eugenics* 7.2 (1936), pp. 179–188.

[4] D. Fuchsgruber. "Integration of SGDE-based Classification into the SG++ Datamining Pipeline." In: (2018).

[5] J. Garcke. "A dimension adaptive sparse grid combination technique for machine learning." In: *ANZIAM Journal* 48 (2007), pp. 725–740.

[6] M. Hegland, G. Hooker, and S. Roberts. "Finite element thin plate splines in density estimation." In: *ANZIAM Journal* 42 (2009), pp. 712–734.

[7] H. Möller. "Dimension-wise Spatial-adaptive Refinement with the Sparse Grid Combination Technique." In: (2018).

[8] M. Obersteiner and H.-J. Bungartz. "A Spatially Adaptive Sparse Grid Combination Technique for Numerical Quadrature." In: (2019).

[9] 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.

[10] B. Peherstorfer. "Model order reduction of parametrized systems with sparse grid learning techniques." PhD thesis. Technische Universität München, 2013.

[11] D. M. Pflüger. "Spatially adaptive sparse grids for high-dimensional problems." PhD thesis. Technische Universität München, 2010.

A. Appendix

The Appendix contains the configuration JSON-files and the results of the tests as well as the hardware specifications of the laptop they were run on.

A.1. Hardware Specifications

CPU	Intel(R) Core(TM) i7-5500U CPU @ 2.40GHz
RAM	2 x 4GB SODIMM DDR3 Synchronous 1600 MHz

A.2. Test Results and Configurations

The results contain the average accuracy and time for the update step around certain numbers of grid points. l_err and u_err denote the maximal upper and lower variance to the value to their left.

A.2.1. The Circles Data Set

Old Implementation

```
{
        "dataSource": {
                "filePath": "../../datasets/niglz/Circles500.arff",
                "hasTargets" :true,
                "batchSize" :0,
                "validationPortion" :0.5,
                "epochs" :140,
                "shuffling" :"random",
                "randomSeed" :-1
        },
        "scorer": {
                "metric": "Accuracy"
        },
        "fitter": {
```

```
15              "type": "classification",
16              "gridConfig": {
17                      "generalGridType": "regular",
18                      "gridType": "linear",
19                      "level": 1
20              },
21              "adaptivityConfig": {
22                              "numRefinements": 120,
23                              "threshold": 0.001,
24                              "maxLevelType": false,
25                              "noPoints": 10,
26                              "refinementIndicator" :"DataBased",
27                              "errorBasedRefinement" :true,
28                              "errorMinInterval" :1,
29                              "errorBufferSize" :2,
30                              "errorConvergenceThreshold" :0.001
31              },
32              "regularizationConfig": {
33                      "lambda": 1e-2
34              },
35              "densityEstimationConfig" :{
36                      "densityEstimationType" :"decomposition",
37                      "matrixDecompositionType" :"Chol"
38              },
39              "learner" :{
40                      "usePrior" :true,
41                      "beta" :1.0
42              }
43      }
44  }
```

points	acc val	l_err	u_err	acc train	l_err	u_err	time	l_err	u_err
7500	0.9407	0.0273	0.0367	0.8721	0.04388	0.04812	166.2	26.77	29.23
6500	0.9441	0.03188	0.03612	0.8793	0.05271	0.05129	132.9	28.12	25.88
5500	0.9448	0.03121	0.03679	0.8818	0.05016	0.04984	96.47	23.53	18.47
4500	0.9445	0.03152	0.03648	0.881	0.04698	0.04902	66.22	17.78	15.22
3500	0.9436	0.02838	0.03562	0.8796	0.04437	0.05163	42.85	12.15	9.849
2500	0.9434	0.0286	0.0394	0.8791	0.04889	0.05111	25.82	8.177	6.823
2050	0.976	1.11e-16	1.11e-16	0.912	1.11e-16	1.11e-16	21.0	0.0	0.0
1950	0.9435	0.02852	0.03548	0.8817	0.04235	0.05365	18.52	1.478	2.522
1850	0.9418	0.03423	0.03377	0.8772	0.04682	0.05318	17.15	0.8526	2.147
1750	0.941	0.03102	0.04098	0.8788	0.04523	0.04677	16.0	2.0	2.0
1650	0.9396	0.03243	0.03957	0.8773	0.04668	0.04932	15.13	0.8738	2.126
1550	0.9418	0.03018	0.04182	0.8798	0.04415	0.04785	14.34	1.658	2.342
1450	0.9401	0.03192	0.04408	0.8776	0.04244	0.04956	13.85	1.147	1.853
1350	0.9397	0.03227	0.04373	0.8786	0.04542	0.05458	12.92	1.077	1.923
1250	0.9413	0.02673	0.03727	0.8806	0.04337	0.04863	12.21	0.7921	2.208
1150	0.9405	0.03147	0.04053	0.8789	0.0451	0.0469	11.53	1.467	1.533
1050	0.9381	0.02991	0.04209	0.8754	0.04855	0.04745	10.81	1.193	1.807
950	0.938	0.034	0.046	0.8797	0.04428	0.05172	10.2	0.7982	1.202
850	0.9369	0.02707	0.03693	0.8789	0.04514	0.04686	9.722	1.278	1.722
750	0.9359	0.02413	0.03987	0.8802	0.04384	0.04816	9.369	0.6311	1.369
650	0.9347	0.02533	0.04267	0.8795	0.0405	0.0475	9.133	0.8667	1.133
550	0.9316	0.02439	0.04361	0.8783	0.04172	0.05428	8.64	1.36	1.64
450	0.9269	0.02513	0.03487	0.8752	0.04485	0.04715	8.092	0.9076	1.092
350	0.9215	0.02649	0.04551	0.8727	0.0433	0.0487	7.67	1.33	1.67
250	0.9045	0.03946	0.04054	0.8589	0.05308	0.05492	7.12	0.88	1.12
150	0.8884	0.03564	0.03236	0.8455	0.04648	0.03752	6.09	0.91	1.09
50	0.6094	0.1026	0.07742	0.5392	0.1248	0.0952	4.86	1.14	0.86
15	0.5284	0.02756	0.02444	0.4791	0.02492	0.02308	4.0	1.0	1.0

New Implementation

```
{
        "dataSource": {
                "filePath": "../../datasets/niglz/Circles500.arff",
                "hasTargets" :true,
                "batchSize" :0,
                "validationPortion" :0.5,
                "epochs" :60,
```

```
                "shuffling" :"random",
                "randomSeed" :-1
        },
        "scorer": {
                "metric": "Accuracy"
        },
        "fitter": {
                "type": "classification",
                "gridConfig": {
                        "generalGridType": "component",
                        "gridType": "linear",
                        "level": 1
                },
                "adaptivityConfig": {
                                "numRefinements": 60,
                                "threshold": 0.001,
                                "maxLevelType": false,
                                "noPoints": 10,
                                "refinementIndicator" :"DataBased",
                                "errorBasedRefinement" :true,
                                "errorMinInterval" :1,
                                "errorBufferSize" :2,
                                "errorConvergenceThreshold" :0.001
                },
                "regularizationConfig": {
                        "lambda": 1e-2
                },
                "densityEstimationConfig" :{
                        "densityEstimationType" :"decomposition",
                        "matrixDecompositionType" :"Chol"
                },
                "learner" :{
                        "usePrior" :true,
                        "beta" :1.0
                }
        }
}
```

points	acc val	l_err	u_err	acc train	l_err	u_err	time	l_err	u_err
8500	0.9719	0.008065	0.01194	0.9231	0.02091	0.02709	33.62	76.38	33.62
7500	0.971	0.009043	0.006957	0.929	0.01504	0.02496	33.14	55.86	33.14
6500	0.9662	0.009771	0.01023	0.9199	0.02409	0.02791	68.8	117.2	68.8
5500	0.9615	0.01048	0.01352	0.9175	0.02653	0.02947	29.71	52.29	29.71
4500	0.955	0.013	0.011	0.908	0.028	0.024	22.38	25.62	22.38
3500	0.9578	0.01425	0.02175	0.9126	0.03143	0.04057	17.19	37.81	17.19
2500	0.9366	0.02736	0.02464	0.8934	0.04662	0.04138	10.79	20.21	10.79
1950	0.9207	0.01133	0.01667	0.8757	0.03633	0.03967	3.917	4.083	3.917
1850	0.9214	0.02655	0.02145	0.8809	0.02715	0.03285	11.94	14.06	11.94
1750	0.9392	0.0128	0.0352	0.8776	0.0264	0.0256	9.1	10.9	9.1
1650	0.9351	0.02088	0.01912	0.8958	0.03624	0.03176	11.36	12.64	11.36
1450	0.9253	0.01867	0.02133	0.8947	0.02933	0.03467	10.17	10.83	10.17
1350	0.9305	0.02545	0.01855	0.8831	0.03289	0.02711	10.21	12.79	10.21
1150	0.9011	0.01091	0.009091	0.8865	0.03345	0.04655	4.591	9.409	4.591
1050	0.8736	0.02639	0.02961	0.7825	0.05752	0.03848	4.739	11.26	4.739
950	0.8629	0.02511	0.01889	0.779	0.06902	0.03898	4.5	5.5	4.5
850	0.8639	0.03615	0.02785	0.7847	0.05933	0.04067	3.5	6.5	3.5
750	0.8609	0.0311	0.0249	0.7968	0.04722	0.04478	8.776	11.22	8.776
650	0.876	0.028	0.028	0.7716	0.04836	0.03564	7.364	8.636	7.364
550	0.8272	0.04078	0.04722	0.7511	0.06885	0.04315	7.626	11.37	7.626
450	0.833	0.015	0.021	0.736	0.024	0.028	7.375	8.625	7.375
350	0.7968	0.03515	0.04085	0.7488	0.06317	0.06083	2.741	5.259	2.741
250	0.8067	0.02928	0.04672	0.7635	0.05654	0.07546	5.131	13.87	5.131
150	0.7471	0.06894	0.07106	0.7051	0.07887	0.1131	4.679	9.321	4.679
50	0.6204	0.1156	0.08837	0.5561	0.1239	0.0921	3.441	9.559	3.441
5	0.5312	0.02484	0.02716	0.4781	0.03388	0.03012	4.915	0.085	4.915

A.2.2. The Funny Chess Data Set

Old Implementation

```
{
        "dataSource": {
                "filePath": "../../datasets/niglz/funnychess.arff",
                "hasTargets" :true,
                "batchSize" :0,
                "validationPortion" :0.5,
                "epochs" :140,
```

```
8               "shuffling" :"random",
9               "randomSeed" :-1
10      },
11      "scorer": {
12              "metric": "Accuracy"
13      },
14      "fitter": {
15              "type": "classification",
16              "gridConfig": {
17                      "generalGridType": "regular",
18                      "gridType": "linear",
19                      "level": 1
20              },
21              "adaptivityConfig": {
22                              "numRefinements": 120,
23                              "threshold": 0.001,
24                              "maxLevelType": false,
25                              "noPoints": 10,
26                              "refinementIndicator" :"DataBased",
27                              "errorBasedRefinement" :true,
28                              "errorMinInterval" :1,
29                              "errorBufferSize" :2,
30                              "errorConvergenceThreshold" :0.001
31              },
32              "regularizationConfig": {
33                      "lambda": 1e-2
34              },
35              "densityEstimationConfig" :{
36                      "densityEstimationType" :"decomposition",
37                      "matrixDecompositionType" :"Chol"
38              },
39              "learner" :{
40                      "usePrior" :true,
41                      "beta" :1.0
42              }
43      }
44  }
```

points	acc val	l_err	u_err	acc train	l_err	u_err	time	l_err	u_err
6500	0.9619	0.01888	0.02472	0.9555	0.0245	0.0303	91.73	8.267	9.733
5500	0.9589	0.02347	0.03173	0.9518	0.02537	0.03223	78.6	15.4	12.6
4500	0.9568	0.02321	0.03439	0.9498	0.02623	0.03537	59.83	10.17	9.832
3500	0.9546	0.02139	0.03581	0.9474	0.02623	0.03897	44.43	8.573	7.427
2500	0.9514	0.02104	0.03736	0.9436	0.02719	0.04081	31.02	5.981	5.019
2050	0.9589	0.01594	0.01966	0.9478	0.01697	0.02423	26.29	0.7143	1.286
1950	0.9474	0.02065	0.04055	0.9392	0.028	0.0512	25.23	0.7664	1.234
1850	0.9451	0.02335	0.04145	0.9364	0.03156	0.05164	24.27	0.73	2.27
1750	0.9439	0.02887	0.04513	0.9349	0.03668	0.05052	23.4	1.6	1.4
1650	0.9416	0.02835	0.04445	0.9326	0.03185	0.05055	22.81	1.19	1.81
1550	0.9373	0.02874	0.04086	0.9283	0.03454	0.04466	22.04	0.96	1.04
1450	0.9345	0.03108	0.03932	0.9251	0.0337	0.0435	21.31	0.6863	1.314
1350	0.9283	0.03653	0.03907	0.9183	0.0389	0.0451	20.43	0.5741	1.426
1250	0.9201	0.04352	0.05088	0.9093	0.04794	0.04926	19.59	1.411	0.5893
1150	0.9115	0.05051	0.05229	0.9004	0.05763	0.05717	18.8	1.197	0.8033
1050	0.9039	0.0533	0.0435	0.8921	0.06194	0.04646	18.78	0.22	0.78
950	0.8886	0.04181	0.03619	0.8764	0.05118	0.04322	18.01	0.99	1.01
850	0.8705	0.0347	0.0345	0.8588	0.03637	0.03203	17.35	0.65	0.35
750	0.8583	0.02214	0.02546	0.8446	0.02575	0.02705	16.62	0.38	0.62
650	0.8436	0.02436	0.02604	0.8274	0.02703	0.02937	15.93	1.066	0.934
550	0.8126	0.03099	0.03501	0.7944	0.03875	0.04045	15.12	0.8763	1.124
450	0.7774	0.02817	0.03823	0.7555	0.03007	0.03713	13.85	1.15	0.85
350	0.7568	0.02482	0.03118	0.7369	0.03469	0.03691	12.92	1.08	0.92
250	0.6169	0.07754	0.06886	0.5874	0.06903	0.07257	11.46	0.54	0.46
150	0.5108	0.02316	0.02364	0.4951	0.02206	0.02354	9.99	0.01	0.99
50	0.5014	0.02616	0.02224	0.4995	0.01568	0.02312	7.97	1.03	0.97

New Implementation

```
{
        "dataSource": {
                "filePath": "../../datasets/niglz/funnychess.arff",
                "hasTargets" :true,
                "batchSize" :0,
                "validationPortion" :0.5,
                "epochs" :60,
                "shuffling" :"random",
                "randomSeed" :-1
```

```
        },
        "scorer": {
                "metric": "Accuracy"
        },
        "fitter": {
                "type": "classification",
                "gridConfig": {
                        "generalGridType": "component",
                        "gridType": "linear",
                        "level": 1
                },
                "adaptivityConfig": {
                                "numRefinements": 60,
                                "threshold": 0.001,
                                "maxLevelType": false,
                                "noPoints": 10,
                                "refinementIndicator" :"DataBased",
                                "errorBasedRefinement" :true,
                                "errorMinInterval" :1,
                                "errorBufferSize" :2,
                                "errorConvergenceThreshold" :0.001
                },
                "regularizationConfig": {
                        "lambda": 1e-2
                },
                "densityEstimationConfig" :{
                        "densityEstimationType" :"decomposition",
                        "matrixDecompositionType" :"Chol"
                },
                "learner" :{
                        "usePrior" :true,
                        "beta" :1.0
                }
        }
}
```

points	acc val	l_err	u_err	acc train	l_err	u_err	time	l_err	u_err
11500	0.9978	0.001372	0.001828	0.9954	0.002184	0.002616	181.5	201.5	181.5
10500	0.9969	0.001533	0.001667	0.9952	0.002433	0.003167	118.3	123.7	118.3
9500	0.9976	0.001645	0.002355	0.9954	0.001806	0.002194	70.51	97.49	70.51
8500	0.9976	0.0016	0.0024	0.9953	0.001928	0.002072	39.76	39.24	39.76
7500	0.9975	0.001691	0.002309	0.9954	0.001826	0.002174	34.44	36.56	34.44
6500	0.9973	0.001948	0.002452	0.9952	0.002036	0.002764	32.6	54.4	32.6
4500	0.9971	0.00167	0.00273	0.9948	0.002374	0.003226	28.61	48.39	28.61
3500	0.9961	0.00188	0.00252	0.9936	0.002762	0.003238	13.09	25.91	13.09
2500	0.9831	0.01409	0.01791	0.98	0.01557	0.01803	15.53	30.47	15.53
1750	0.939	0.01176	0.01544	0.9325	0.01552	0.02008	4.33	10.67	4.33
1650	0.952	0.01802	0.01918	0.9427	0.0221	0.0235	15.62	16.38	15.62
1550	0.9405	0.01231	0.01369	0.9368	0.01156	0.02044	12.33	13.67	12.33
1450	0.9496	0.01763	0.02997	0.9424	0.02323	0.02877	10.91	11.09	10.91
1350	0.9425	0.007533	0.01047	0.9354	0.005	0.0086	7.333	8.667	7.333
1250	0.9538	0.01423	0.01577	0.9457	0.02031	0.02209	8.395	9.605	8.395
1150	0.9527	0.01565	0.03035	0.946	0.01996	0.02564	6.355	6.645	6.355
1050	0.9071	0.02971	0.04469	0.8941	0.03234	0.05886	18.55	20.45	18.55
850	0.8426	0.06178	0.09022	0.8249	0.06913	0.09527	5.685	6.315	5.685
750	0.7056	0.07201	0.06279	0.6837	0.07793	0.07167	13.39	14.61	13.39
550	0.4612	0.05119	0.03761	0.4458	0.04817	0.04103	12.16	13.84	12.16
350	0.4993	0.02949	0.03011	0.4861	0.03231	0.03089	6.975	9.025	6.975
250	0.4937	0.03026	0.02974	0.4806	0.02665	0.02455	8.895	10.11	8.895
150	0.4927	0.03729	0.03431	0.4822	0.04056	0.03624	8.408	17.59	8.408
50	0.4976	0.00878	0.00722	0.5024	0.00762	0.00838	5.15	10.85	5.15
5	0.4976	0.00878	0.00722	0.5024	0.00762	0.00838	6.28	2.72	6.28

A.2.3. The Iris Data Set

Old Implementation

```
{
        "dataSource": {
                "filePath": "../../datasets/niglz/irisnorm.arff",
                "fileType": "arff",
                "hasTargets" :true,
                "batchSize" :0,
                "epochs": 40,
                "validationPortion" :0.5,
```

```
9               "shuffling" :"random",
10              "randomSeed": -1,
11              "readinClasses": [1,2,3]
12      },
13      "scorer": {
14              "metric": "Accuracy"
15      },
16      "fitter": {
17              "type": "classification",
18              "gridConfig": {
19                      "generalGridType": "regular",
20                      "gridType": "linear",
21                      "level": 1
22              },
23              "adaptivityConfig": {
24                              "numRefinements": 40,
25                              "threshold": 0.001,
26                              "maxLevelType": false,
27                              "noPoints": 10,
28                              "refinementIndicator" :"DataBased",
29                              "errorBasedRefinement" :true,
30                              "errorMinInterval" :1,
31                              "errorBufferSize" :2,
32                              "errorConvergenceThreshold" :0.001
33              },
34              "regularizationConfig": {
35                      "lambda": 1e-2
36              },
37              "densityEstimationConfig" :{
38                      "densityEstimationType" :"decomposition",
39                      "decomposition": "Chol"
40              },
41              "learner" :{
42                      "usePrior" :true,
43                      "beta" :1.0
44              }
45      }
46  }
```

points	acc val	l_err	u_err	acc train	l_err	u_err	time	l_err	u_err
9500	0.9589	1.11e-16	1.11e-16	0.8767	0.0	0.0	207.0	0.0	0.0
8500	0.9401	0.01884	0.02226	0.9024	0.02911	0.02569	186.4	32.62	36.38
7500	0.9416	0.01727	0.03752	0.9083	0.05063	0.04526	154.4	24.61	15.39
6500	0.9462	0.02643	0.04206	0.8993	0.05962	0.03627	127.0	29.04	25.96
5500	0.9476	0.02499	0.0298	0.8998	0.04538	0.03681	101.3	22.73	18.27
4500	0.9357	0.03687	0.04532	0.9127	0.04622	0.04967	76.49	17.51	16.49
3500	0.9333	0.03926	0.05663	0.9161	0.05654	0.05305	59.73	12.27	11.73
2500	0.9347	0.03791	0.04429	0.9145	0.05811	0.03778	46.64	9.358	9.642
1950	0.9227	0.02247	0.03233	0.9315	0.0411	0.0274	39.92	3.08	5.92
1850	0.925	0.06135	0.03454	0.9136	0.03157	0.03693	38.39	4.609	5.391
1750	0.9358	0.0368	0.04539	0.9143	0.05829	0.0513	37.65	5.353	5.647
1650	0.9264	0.03249	0.03601	0.9186	0.04031	0.04188	35.54	5.457	5.543
1550	0.9292	0.02968	0.05251	0.9155	0.05708	0.03881	33.71	4.292	4.708
1450	0.9385	0.04782	0.04807	0.9101	0.06249	0.0471	33.96	6.035	5.965
1350	0.9211	0.03783	0.03066	0.9217	0.02348	0.03131	32.38	5.619	4.381
1250	0.9228	0.03611	0.04608	0.9278	0.04483	0.03736	33.0	2.0	6.0
1150	0.932	0.04057	0.04162	0.912	0.06059	0.049	31.13	4.872	3.128
1050	0.9241	0.03477	0.04742	0.9168	0.04215	0.04004	31.54	2.462	5.538
950	0.9131	0.03206	0.03643	0.9303	0.04226	0.03993	28.51	3.489	3.511
850	0.9408	0.03178	0.05041	0.8984	0.04685	0.03534	26.16	2.84	3.16
750	0.9123	0.03288	0.04931	0.9299	0.04274	0.03945	24.2	2.8	2.2
550	0.9237	0.03521	0.04699	0.9142	0.04466	0.05123	20.3	2.7	2.3
350	0.9166	0.04233	0.03986	0.9048	0.04041	0.04178	14.54	2.46	1.54
150	0.8582	0.07329	0.1048	0.8459	0.07192	0.09247	8.53	1.47	1.53
50	0.4251	0.05438	0.05521	0.2737	0.08247	0.06822	5.92	1.08	0.92

New Implementation

```
{
        "dataSource": {
                "filePath": "../../datasets/niglz/irisnorm.arff",
                "fileType": "arff",
                "hasTargets" :true,
                "batchSize" :0,
                "epochs": 50,
                "validationPortion" :0.5,
                "shuffling" :"random",
                "randomSeed": -1,
```

```
                "readinClasses": [1,2,3]
        },
        "scorer": {
                "metric": "Accuracy"
        },
        "fitter": {
                "type": "classification",
                "gridConfig": {
                        "generalGridType": "component",
                        "gridType": "linear",
                        "level": 1
                },
                "adaptivityConfig": {
                                "numRefinements": 50,
                                "threshold": 0.001,
                                "maxLevelType": false,
                                "noPoints": 10,
                                "refinementIndicator" :"DataBased",
                                "errorBasedRefinement" :true,
                                "errorMinInterval" :1,
                                "errorBufferSize" :2,
                                "errorConvergenceThreshold" :0.001
                },
                "regularizationConfig": {
                        "lambda": 1e-2
                },
                "densityEstimationConfig" :{
                        "densityEstimationType" :"decomposition",
                        "decomposition": "Chol"
                },
                "learner" :{
                        "usePrior" :true,
                        "beta" :1.0
                }
        }
}
```

points	acc val	l_err	u_err	acc train	l_err	u_err	time	l_err	u_err
2500	0.9263	0.0463	0.0359	0.9024	0.04277	0.05312	11.54	20.46	11.54
1950	0.9296	0.04299	0.03921	0.9029	0.04228	0.05361	12.89	24.11	12.89
1850	0.9295	0.0568	0.03909	0.9024	0.04277	0.05312	10.02	16.98	10.02
1750	0.9245	0.04813	0.03406	0.9028	0.04245	0.05344	9.928	20.07	9.928
1650	0.9284	0.04418	0.03801	0.8949	0.03664	0.03185	12.07	24.93	12.07
1550	0.9212	0.03767	0.04452	0.9059	0.03926	0.05663	7.625	20.38	7.625
1450	0.9252	0.04738	0.04851	0.9005	0.04469	0.0512	8.426	14.57	8.426
1350	0.9291	0.04351	0.03868	0.8961	0.04915	0.04674	8.618	22.38	8.618
1250	0.9136	0.03163	0.1191	0.9141	0.04483	0.06476	11.81	34.19	11.81
1150	0.9255	0.04707	0.03512	0.8976	0.04759	0.07569	19.11	26.89	19.11
1050	0.9118	0.06079	0.0625	0.9092	0.03596	0.04623	14.34	25.66	14.34
950	0.9215	0.03742	0.04477	0.9011	0.04406	0.05183	11.98	17.02	11.98
850	0.9111	0.06155	0.07544	0.8842	0.03359	0.0623	9.767	18.23	9.767
750	0.9107	0.04817	0.04772	0.8818	0.04974	0.05985	10.73	20.27	10.73
650	0.9175	0.05509	0.0545	0.887	0.04448	0.05141	11.1	15.9	11.1
550	0.9139	0.04501	0.05088	0.8873	0.04419	0.0517	11.05	18.95	11.05
450	0.9175	0.05514	0.04075	0.8955	0.03596	0.05993	8.275	18.73	8.275
350	0.9186	0.04034	0.05555	0.8981	0.04712	0.04877	7.18	9.82	7.18
250	0.9158	0.05678	0.06651	0.8943	0.05089	0.045	11.65	25.35	11.65
150	0.8842	0.08836	0.1445	0.8704	0.07479	0.1444	8.845	13.15	8.845
50	0.8868	0.07215	0.1059	0.8623	0.06922	0.1363	9.003	22.0	9.003
5	0.4307	0.06247	0.04712	0.2823	0.07384	0.07685	5.97	3.03	5.97

A.2.4. The DR10 Data Set

Old Implementation

```
{
        "dataSource": {
                "filePath": "../../datasets/niglz/dr10_train_equal.arff",
                "hasTargets" :true,
                "batchSize" :0,
                "validationPortion" :0.5,
                "epochs" :225,
                "shuffling" :"random",
                "randomSeed" :-1,
                "readinCutoff": -1,
                "readinClasses": [0,1]
```

```
        },
        "scorer": {
                "metric": "Accuracy"
        },
        "fitter": {
                "type": "classification",
                "gridConfig": {
                        "generalGridType": "regular",
                        "gridType": "linear",
                        "level": 1
                },
                "adaptivityConfig": {
                                "numRefinements": 2000,
                                "threshold": 0.001,
                                "maxLevelType": false,
                                "noPoints": 10,
                                "refinementIndicator" :"DataBased",
                                "errorBasedRefinement" :true,
                                "errorMinInterval" :1,
                                "errorBufferSize" :2,
                                "errorConvergenceThreshold" :0.001
                },
                "regularizationConfig": {
                        "lambda": 1e-7
                },
                "densityEstimationConfig" :{
                        "densityEstimationType" :"decomposition",
                        "decomposition": "Chol"
                },
                "learner" :{
                        "usePrior" :true,
                        "beta" :1.0
                }
        }
}
```

points	acc val	l_err	u_err	acc train	l_err	u_err	time	l_err	u_err
15500	0.6771	0.2016	0.2016	0.6749	0.1997	0.1997	2.072e+03	87.5	87.5
14500	0.7855	0.1051	0.2988	0.783	0.1026	0.2924	1.977e+03	135.3	141.7
13500	0.7118	0.1749	0.2419	0.7095	0.1719	0.2408	1.825e+03	114.2	151.8
12500	0.752	0.1378	0.287	0.7502	0.1351	0.286	1.821e+03	190.2	178.8
11500	0.7628	0.1236	0.2966	0.7607	0.1223	0.2949	1.705e+03	107.2	195.8
10500	0.6881	0.2013	0.2032	0.6867	0.1992	0.2021	1.596e+03	100.6	101.4
9500	0.7796	0.1109	0.2893	0.778	0.1091	0.2869	1.456e+03	85.6	151.4
8500	0.7259	0.1618	0.2544	0.7248	0.1605	0.2531	1.338e+03	71.43	82.57
7500	0.7671	0.1262	0.2806	0.7656	0.1243	0.2793	1.163e+03	32.09	30.91
6500	0.6772	0.2125	0.2256	0.6766	0.2113	0.2253	1.121e+03	43.88	57.12
5500	0.7208	0.1688	0.2354	0.7209	0.1669	0.2374	1e+03	80.67	73.33
4500	0.7565	0.1321	0.2911	0.7557	0.1317	0.2914	859.5	90.47	89.53
3500	0.8125	0.07271	0.3201	0.8121	0.07262	0.3199	707.2	80.77	40.23
2500	0.7254	0.156	0.2504	0.7256	0.156	0.2512	609.5	62.55	110.5
1750	0.8224	0.06042	0.1125	0.8223	0.05881	0.1139	499.4	8.6	4.4
1650	0.7642	0.1089	0.2971	0.7639	0.109	0.2974	446.0	4.0	3.0
1350	0.7392	0.1093	0.2559	0.7396	0.1073	0.253	418.6	2.4	1.6
1250	0.7057	0.1349	0.2118	0.7058	0.1325	0.2113	339.0	13.0	5.0
1050	0.6642	0.04515	0.05236	0.6637	0.04593	0.05275	293.7	3.333	2.667
950	0.6387	0.1427	0.1427	0.6384	0.1423	0.1423	292.5	1.5	1.5
850	0.7152	0.1013	0.2123	0.7164	0.1021	0.213	245.4	6.6	7.4
650	0.7695	0.04093	0.03404	0.7678	0.04123	0.034	179.0	3.0	4.0
550	0.7391	0.08188	0.1178	0.7381	0.0816	0.1168	132.7	1.333	0.6667
450	0.6547	0.1657	0.1657	0.6552	0.1654	0.1654	132.5	0.5	0.5
350	0.6799	0.132	0.2198	0.6798	0.131	0.2174	94.0	21.0	8.0
250	0.7277	0.07588	0.09911	0.7273	0.0729	0.09525	54.4	10.6	5.4
50	0.5901	0.1499	0.09013	0.5902	0.1502	0.09031	28.0	6.0	7.0

New Implementation

```
{
        "dataSource": {
                "filePath": "../../datasets/niglz/dr10_train_equal.arff",
                "hasTargets" :true,
                "batchSize" :0,
                "validationPortion" :0.5,
                "epochs" :175,
                "shuffling" :"random",
```

```
9               "randomSeed" :-1,
10              "readinCutoff": -1,
11              "readinClasses": [0,1]
12      },
13      "scorer": {
14              "metric": "Accuracy"
15      },
16      "fitter": {
17              "type": "classification",
18              "gridConfig": {
19                      "generalGridType": "component",
20                      "gridType": "linear",
21                      "level": 1
22              },
23              "adaptivityConfig": {
24                              "numRefinements": 2000,
25                              "threshold": 0.001,
26                              "maxLevelType": false,
27                              "noPoints": 10,
28                              "refinementIndicator" :"DataBased",
29                              "errorBasedRefinement" :true,
30                              "errorMinInterval" :1,
31                              "errorBufferSize" :2,
32                              "errorConvergenceThreshold" :0.001
33              },
34              "regularizationConfig": {
35                      "lambda": 1e-7
36              },
37              "densityEstimationConfig" :{
38                      "densityEstimationType" :"decomposition",
39                      "decomposition": "Chol"
40              },
41              "learner" :{
42                      "usePrior" :true,
43                      "beta" :1.0
44              }
45      }
46  }
```

points	acc val	l_err	u_err	acc train	l_err	u_err	time	l_err	u_err
23500	0.665	0.0	0.0	0.6665	1.11e-16	1.11e-16	118.0	0.0	0.0
22500	0.6568	1.11e-16	1.11e-16	0.6592	2.22e-16	2.22e-16	202.5	200.5	200.5
21500	0.7394	1.11e-16	1.11e-16	0.7406	1.11e-16	1.11e-16	64.5	62.5	62.5
20500	0.7634	0.05022	0.06842	0.765	0.04965	0.07208	75.67	148.3	74.67
19500	0.7559	0.07047	0.1471	0.757	0.07004	0.1483	57.92	77.08	56.92
18500	0.7564	0.08418	0.1685	0.7588	0.08278	0.1705	92.62	124.4	91.62
17500	0.726	0.1077	0.1702	0.7273	0.1078	0.17	48.07	73.93	47.07
16500	0.7464	0.08221	0.1847	0.7476	0.07732	0.1859	35.73	166.3	34.73
15500	0.737	0.09441	0.2043	0.738	0.09044	0.2035	58.9	86.1	57.9
14500	0.7149	0.101	0.1742	0.7157	0.09668	0.1733	58.48	222.5	57.48
13500	0.7595	0.06185	0.08226	0.7602	0.06403	0.08701	75.0	128.0	74.0
12500	0.7701	0.06564	0.05074	0.7706	0.06339	0.05559	46.05	85.95	45.05
11500	0.7676	0.05635	0.03809	0.7686	0.05443	0.04107	60.95	82.05	59.95
10500	0.7603	0.06543	0.03625	0.7619	0.06293	0.0408	67.25	76.75	66.25
9500	0.7601	0.06574	0.05717	0.7615	0.06352	0.06141	84.3	123.7	83.3
8500	0.7696	0.06925	0.07324	0.7699	0.06854	0.07786	64.0	81.0	63.0
7500	0.7609	0.08124	0.1225	0.762	0.08193	0.1253	56.58	89.42	55.58
6500	0.742	0.09155	0.1069	0.7432	0.09146	0.1104	36.02	69.98	35.02
5500	0.7154	0.1181	0.2154	0.7154	0.1203	0.2158	39.7	104.3	38.7
4500	0.7156	0.09122	0.1217	0.7163	0.08902	0.1192	55.68	137.3	54.68
3500	0.674	0.08631	0.09385	0.6748	0.08988	0.09904	54.65	125.3	53.65
2500	0.6927	0.1293	0.1918	0.6938	0.127	0.1927	45.81	101.2	44.81
1950	0.664	0.1309	0.163	0.6643	0.1312	0.1633	45.0	70.0	42.0
1750	0.7102	0.104	0.1418	0.7108	0.1046	0.1439	48.0	50.0	46.0
1550	0.7313	0.08521	0.1805	0.7312	0.08711	0.1845	28.22	43.78	26.22
1450	0.768	0.06761	0.1773	0.7676	0.06844	0.1815	38.83	46.17	36.83
1250	0.6304	0.1654	0.09269	0.6304	0.1648	0.09531	27.1	25.9	25.1
1150	0.7305	0.09015	0.1836	0.7306	0.09118	0.1869	21.6	39.4	19.6
1050	0.7239	0.08877	0.1846	0.7239	0.09065	0.1876	66.7	179.3	64.7
850	0.7388	0.09379	0.134	0.7396	0.09619	0.1401	62.5	68.5	60.5
650	0.6598	0.05452	0.03432	0.6604	0.05595	0.04146	41.3	43.7	39.3
550	0.5919	0.01745	0.02714	0.5929	0.01503	0.02578	31.1	32.9	29.1
450	0.5937	0.008404	0.009481	0.5949	0.007816	0.01063	37.5	42.5	35.5
350	0.6613	0.07076	0.03247	0.6627	0.06662	0.02828	53.6	56.4	51.6
250	0.6782	0.03164	0.06887	0.6798	0.03335	0.0714	12.32	19.68	10.32
150	0.5934	0.1089	0.09364	0.5944	0.112	0.09443	51.0	108.0	49.0
50	0.5197	0.01039	0.01996	0.5198	0.01069	0.01976	25.4	48.6	23.4
5	0.5001	9.28e-05	0.0001232	0.4999	0.0001178	6.82e-05	51.7	53.3	49.7