DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics: Games Engineering

Embedded SGDE image classification on mobile devices

Subhan-Jamal Sohail

DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics: Games Engineering

Embedded SGDE image classification on mobile devices

Einbettung der SGDE Bild-Klassifikation auf mobilen Geräten

Author:	Subhan-Jamal Sohail
Supervisor:	Prof. Dr. rer. nat. habil. Hans-Joachim Bungartz
Advisor:	Kilian Röhner, M.Sc.
Submission Date:	22.03.2019

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

Munich, 22.03.2019 Subhan-Jamal Sohail

Abstract

This thesis describes the integration of the geometry aware sparse grids into the datamining pipeline of the SG++ framework. The datamining pipeline is used to make SG++ easier available and fasten the process of using the framework. Geometry aware sparse grids have a great use in image classification and therefore this thesis is additionally about creating a mobile application that classifies hand drawn numbers using this kind of sparse grids. For this to be done the trained data had to be exported from SG++ and then imported again into the application. Also the evaluation method for sparse grids was implemented into the application. This new evaluation method is then validated by comparing it to the evaluation method of SG++. The outcome of the validation resulted in a high deviation for random datasets and very low deviation for relevant datasets. The high deviation for random datasets can be neglected, since it has no use case in the user application. Finally the application was tested for numbers 0, 2 and 6 with various relevant datasets to calculate the accuracy of the implementation. The tests have shown that by following certain rules while drawing the numbers into the application an average accuracy of 90% could be observed.

Zusammenfassung

Diese Arbeit befasst sich mit der Integration der geometrisch bewussten Dünngitter in die Datmining Pipeline der SG++ Toolbox. Die Datamining Pipeline hat den Nutzen die Verwendung der SG++ Toolbox zugänglicher und schneller zu gestalten. Geometrisch bewusste Dünngitter haben einen großen Nutzen in der Klassifizierung von Bildern, deswegen befasst sich diese Arbeit auch mit der Entwicklung einer mobilen Anwendung, die handschriftlich gezeichnete Nummern klassifiziert unter Verwendung eines solchen Dünngitters. Um das zu erreichen müssen die trainierten Daten aus SG++ exportiert und wieder in die Anwendung importiert werden. Zudem musste die Auswertungsfunktion der Dünngitter in die Anwendung implementiert werden. Diese neue Auswertungsfunktion wird dann validiert, indem sie mit der Auswertungsfunktion von SG++ verglichen wird. Das Resultat dieser Validierung ergab eine sehr hohe Abweichung für zufällige Datensätze und eine sehr geringe Abweichung für relevante Datensätze. Die hohe Abweichung für zufällige Datensätze kann jedoch vernachlässigt werden, da diese keinen Anwendungsfall in der Benutzeranwendung finden. Schließlich wurde die Anwendung an verschiedenen relevanten Datensätzen für die Nummern 0, 2 und 6 getestet, um die Genauigkeit der Implementation zu berechnen. In den Tests konnte eine durchschnittliche Genauigkeit von 90% beobeachtet werden, unter Einhaltung bestimmter Regeln während dem Zeichnen der Nummern in die Anwendung.

Contents

1 Introduction

The toolbox SG++ was developed by Dirk Pflüger at the chair of Scientific Computing in 2010 [Pfl10] at the Technical University of Munich. This toolbox implements many methods to perform calculation on adaptive sparse grids. The use and the presentation of the toolbox seems rather counter intuitive to outstanding people. To close this gap and develop an easy use case of sparse grids, a mobile application that classifies images using sparse grids was developed during this thesis.

To create such an image classification application we have to adjust the grid that we use. Sparse grids overcome the difficulty of the *curse of dimensionality* by intelligently leaving out gridpoints and thus reducing the amount of gridpoints significantly. Despite this reduction, the amount of gridpoints in a sparse grid for images is still too high to calculate upon.

Therefore the approach of geometry aware sparse grid is introduced [Wae17]. With this approach the amount of gridpoints for an image can be reduced to a reasonable amount by just taking gridpoints into the grid space that full fill some specific geometric relation.

The use of the toolbox was also simplified through a datamining pipeline that creates and evaluates adaptive sparse grids just by setting parameters in a configuration file. In 2018 Fuchsgruber extended the pipeline to support sparse grid density estimation-based classification [Fuc18]. Up to this date however this classification did not support geometry aware sparse grids, therefore they have been implemented in the span this thesis.

In the following the theory knowledge required is discussed in detail in Chapter 2, followed by the implementation and the decisions that had to be made in Chapter 3. This implementation is then evaluated in Chapter 4. Finally we draw a conclusion and define future work to improve the implementation.

2 Theory

This chapter will give a brief introduction to sparse grids and the density estimation classification method used in this thesis. Also geometry aware sparse grids are discussed in this chapter that are used for image classification in the mobile application.

2.1 Sparse Grids

This chapter will be an introduction to sparse grids by deriving them from full grids and extending them to the d-dimensional case through hierarchical basis functions. Also the density estimation classifcation method, which will be used in the implementation, is discussed here.

2.1.1 Full Grid Interpolation

To explain the full grid interpolation of a function f, we consider the function $f : \Omega \rightarrow \mathbb{R}$. We want to evaluate f on arbitrary points, therefore Ω is restricted to a sub volume of $\mathbb{R}^d$. In the following we will also define the function domain Ω as the d-dimensional unit-hypercube, $\Omega := [0, 1]^d$.
Now to build the interpolant u from our function f, we have to discretize Ω through a regular grid. This regular grid is constructed by $2^n - 1$ equidistant grid points x_i with a mesh width of $h_n := 2^{-n}$, with n being the discretization level. The function f is then evaluated and interpolated on these grid points.
Specifically $u(\vec{x})$ is then obtained by defining basis functions $\varphi_i(\vec{x})$ and then combining them as weighted sum on each grid point, resulting in the following equation

$$f(\vec{x}) \approx u(\vec{x}) := \sum_i \alpha_i \varphi_i(\vec{x}). \tag{2.1}$$

This type of interpolation is illustrated in Figure 2.1 for a one-dimensional case, also a comparison to a linear interpolation can be seen.
The choice of the basis function can be crucial for the runtime of computing the interpolation. Therefore suitable basis function are hierarchical constructed basis functions as stated in [Pfl10].

One dimensional hierarchical basis function To obtain such a function, we can start off with the one-dimensional standard hat function.

$$\varphi(x) = max(1 - |x|, 0) \tag{2.2}$$

From the standard function, we can then derive one-dimensional hat basis functions by translation and dilatation [Pfl10].

$$\varphi_{l,i}(x) = \varphi(2^l x - i) \tag{2.3}$$

The basis functions in (2.3) are now dependent on a level l and an index $i, 0 < i < 2^l$. Each of these functions have local support and are centered at a grid point $x_{l,i} = 2^{-l} i$ on which our function f is interpolated on.

$$I_l := \{i \in \mathbb{N} : 1 \leq i \leq 2^l - 1, i \text{ odd}\} \tag{2.4}$$

By defining a hierarchical index set in (2.9) and taking the span of every basis function $\varphi_{l,i}$, we can obtain the set of hierarchical subspaces W_l.

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

The space of piecewise linear functions on a full grid V_n can then be defined by taking the sum of all hierarchical subspaces W_l for a specific level l [Pfl10].

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

As mentioned in formula (2.1) the interpolated function $u(\vec{x}) \in V_n$ can now be calculated by taking our translated and dilated hierarchical basis functions from (2.3) and combining them as weighted sum on each gridpoint.

$$u(x) = \sum_{l \leq n, i \in I_l} \alpha_{l,i} \varphi_{l,i}(x) \tag{2.7}$$

From one-dimension to d-dimensions The transition into higher dimension can be done by defining the basis functions through the tensor product approach and using multi-indices for $\vec{i}$ and $\vec{l}$ that represent level and index of a dimension [Pfl10].

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

From this point on the interpolant is calculated analogously to the previous calculation of the one-dimensional case. We just replace the index i and level l with the multi-indices $\vec{i}$ and $\vec{l}$.

$$I_{\vec{l}} := \{\vec{i} \in \mathbb{N} : 1 \leq i \leq 2^l - 1, i \text{ odd}\}, \tag{2.9}$$

is our new hierarchical index set to define the set of hierarchical subspaces $W_{\vec{l}}$.

$$W_{\vec{l}} := \operatorname{span}\{\varphi_{\vec{l},\vec{i}}(\vec{x}) : \vec{i} \in I_l\} \tag{2.10}$$

Again by taking the sum of all hierarchical subspaces $W_{\vec{l}}$, we get the space of piecewise d-linear functions V_n [Pfl10].

$$V_n = \bigoplus_{l \leq n} W_{\vec{l}} \tag{2.11}$$

As previously the interpolant $u(\vec{x}) \in V_n$ is the sum of all weighted basis functions on every gridpoint.

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

This type of full grid consists of $(2^n - 1)^d$ gridpoints and as Equation (2.12) shows that for the interpolation of the function f, we have to perform an evaluation on every gridpoint. This leads to an asymptotic behavior of $\mathcal{O}(2^{nd})$ and thus encountering *the curse of dimensionality* [Pfl10]. An example for the subspaces is illustrated in Figure 2.2.

2.1.2 Sparse Grid Interpolation

To bypass the *curse of dimensionality* that we get from the full grid interpolation in Section 2.1.1 the amount of subspaces $W_{\vec{l}}$ has to be reduced. Thus subspaces that only pay a small contribution to the overall solution are left out.

$$V_n^{(1)} = \bigoplus_{|\vec{l}|_1 \leq n+d-1} W_{\vec{l}} \tag{2.13}$$

Applying this sparse grid space $V_n^{(1)}$ to the interpolant $u(\vec{x})$ we get

$$u(\vec{x}) = \sum_{|\vec{l}|_1 \leq n+d-1, \vec{i} \in I_{\vec{l}}} \alpha_{\vec{l},\vec{i}} \varphi_{\vec{l},\vec{i}}(\vec{x}). \tag{2.14}$$

With this the number of gridpoints is reduced significantly, while the asymptotic error only gets slightly worse [Pfl10]. In Figure 2.3 an example sparse grid of the level n = 3 can be seen.

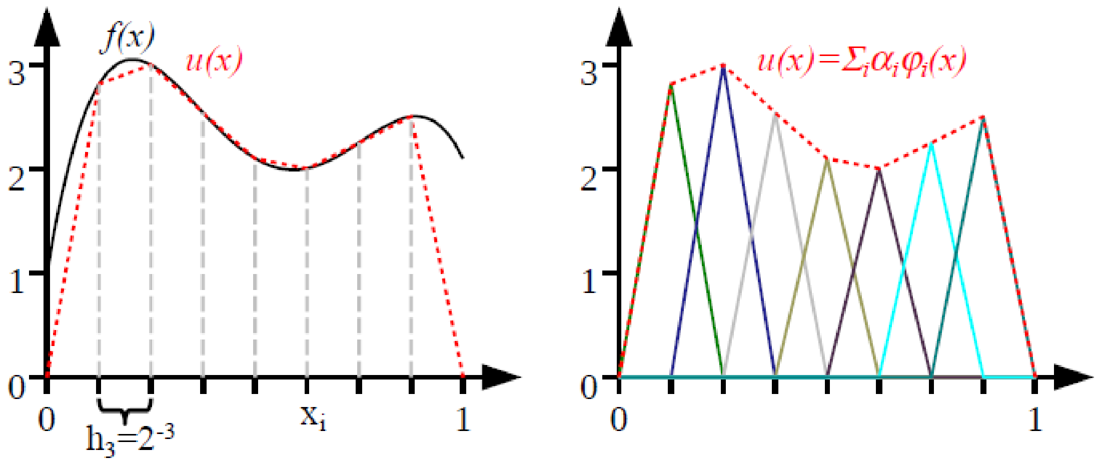


Figure 2.1: Linear interpolation (left) and linear combination from basis hat functions (right). Image from [Pfl10]

Boundary Treatment As for now the basis function were 0 on the border of the grid. To avoid losing information on the boundary the modified linear basis functions are introduced in 2.15 [Pfl10].

$$\varphi_{l,i}(x) := \begin{cases} 1 & \text{if } l = 1 \wedge i = 1 \\ \left.\begin{matrix} 2 - 2^l \cdot x & \text{if } x \in [0, \frac{1}{2^{l-1}}] \\ 0 & \text{else} \end{matrix}\right\} & \text{if } l > 1 \wedge i = 1 \\ \left.\begin{matrix} 2^l \cdot x + 1 - i & \text{if } x \in [1 - \frac{1}{2^{l-1}}, 1] \\ 0 & \text{else} \end{matrix}\right\} & \text{if } l > 1 \wedge i = 2^l - 1 \\ \varphi(x \cdot 2^l - i) & \text{else.} \end{cases} \tag{2.15}$$

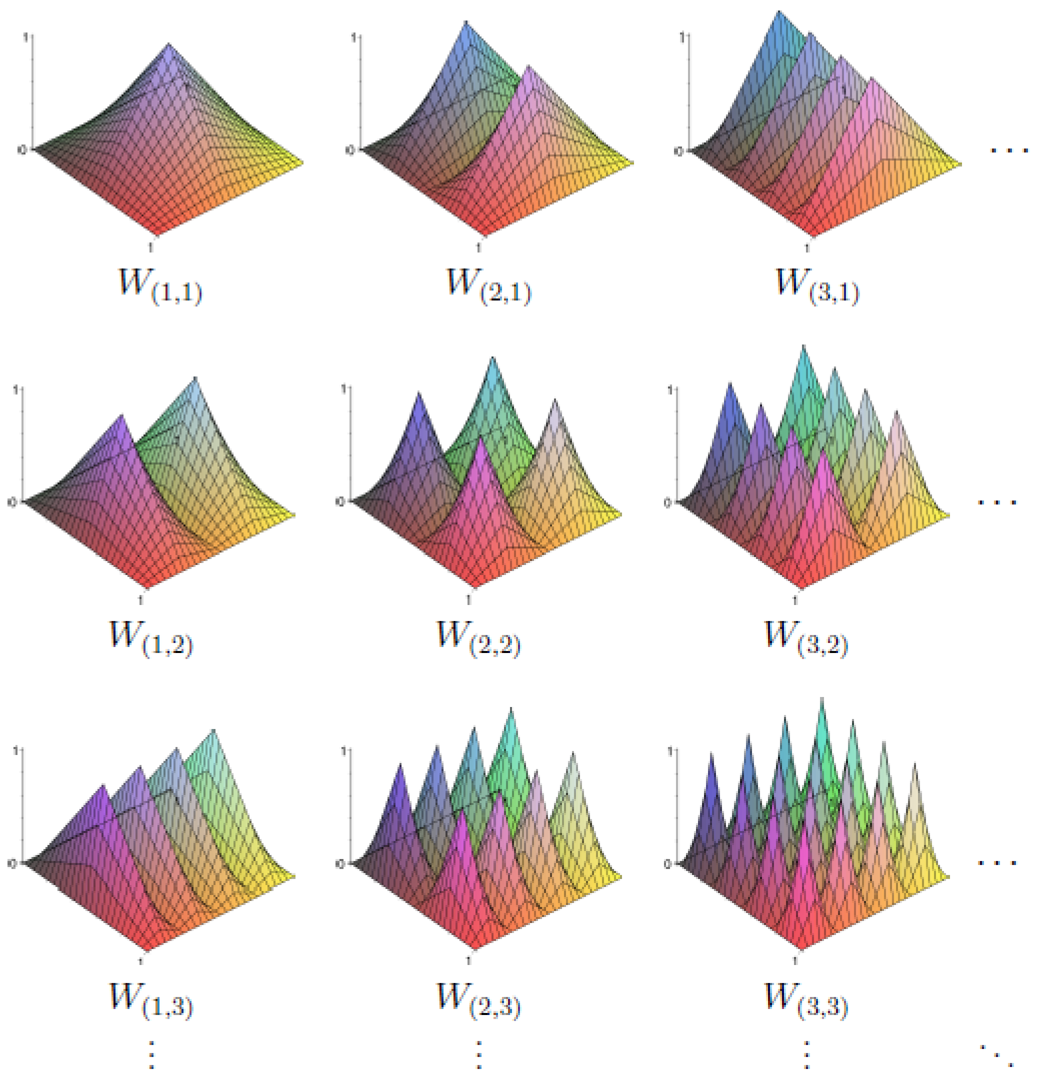


Figure 2.2: Subspaces W_l for $|\vec{l}| \leq 3$. Image from [Pfl10]

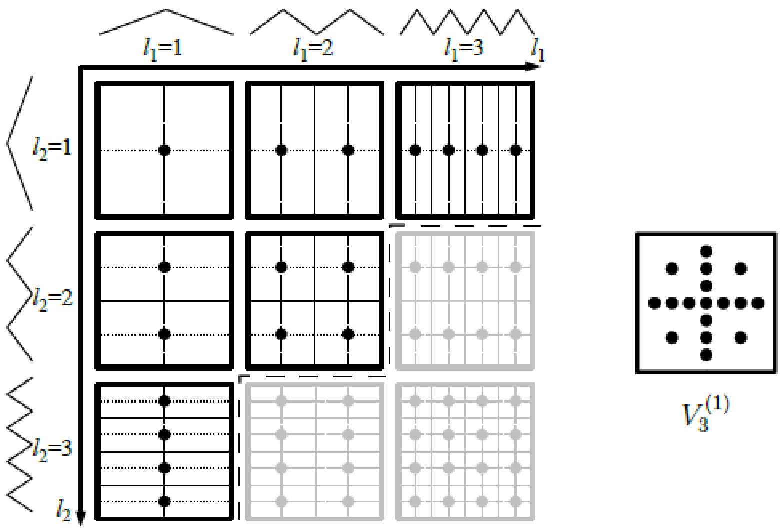


Figure 2.3: Subspaces $W_{\vec{l}}$ of a sparse grid with level l=3 (left) and overlapping all subspaces $W_{\vec{l}}$ results in the sparse grid space $V_3^{(1)}$ (right). Image from [Pfl10]

2.1.3 Sparse Grid Density Estimation

The classification method used in this thesis is the sparse grid based density estimation. We can define the density estimation as following [Peh13].

$$(R + \lambda C)\vec{\alpha} = \vec{b} \tag{2.16}$$

With $R_{i,j} = (\Phi_i, \Phi_j)_{L2}$ representing the underlying sparse grid as L-2 inner product and $C_{i,j} = (\Lambda\Phi_i, \Lambda\Phi_j)_{L2}$ being the regularization matrix with λ as the regularization factor. The notation for the surpluses $\vec{\alpha}$ is the same as previously. Finally the vector $b_i = \frac{1}{M}\sum_{j=1}^{M}\Phi_i(x_j)$ represents the training dataset.
This approach is used since it does not depend on the size of the training dataset, instead it depends on the amount of gripoints N. This is due to R and C being of the size $N \times N$ and the vector b being of the size N [Peh13].
This system of equations can now be solved to calculate the unknown weights $\vec{\alpha}$ of every basis function in a sparse grid, given a sparse grid and a training dataset.

2.2 Geometry Aware Sparse Grids

This is a variant of a sparse grids that chooses gridpoints upon a different property. Usual sparse would leave out those gridpoints that do not contribute a lot to the overall solution, however geometry aware sparse grids only take in gridpoints that have a geometric relation. For specific cases this approach can lead to a significant reduction in gridpoints.
For example considering image classification, in which we can represent an image as a grid. Every pixel in that image represents a separate dimension and the function value of that pixel defines its exact position in a grid.
By looking at the amount of gridpoints of a regular sparse grid for a 8x8x3 image, we can see in Table 2.1 that the growth just after level 2 is already way to high to compute. However now by computing the amount of gridpoints with the geometry aware approach, we can consider a geometric relation in an image by looking at neighbouring pixels. In this case by just looking at the geometric relation of two dimension we get a significant reduction of gridpoints, as we can see in Table 2.1.

	Regular SG vs GASG		
	Level 2	Level 3	Level 4
GASG with DN stencil	385	3009	11969
regular SG	385	74497	9659649

Table 2.1: Comparison in gridsize for a 8x8x3 image. Table data taken from [Wae17]

2.2.1 Interactions

The so called choice of gridpoints upon a geometric relation can also be called interaction. So to apply this approach to a grid, we will define our geometry aware sparse grid space V^T to only take in subspaces W_l that model an interaction T [Wae17].

$$V^T = \bigoplus_{W_l \in V^{(1)} \wedge \zeta(\vec{l})} W_l \tag{2.17}$$

The function $\zeta(\vec{l})$ here, is a function that returns the modeled interactions.

2.2.2 Stencil

The stencil is the property on which interactions get modeled.
For the example of image classification there are various stencils, including the direct

neighbour **DN** or the diagonals **DNDIAG** stencil. The impact on the gridsize of these stencils compared to a sparse regular grid in plotted in Figure 2.5. In this figure we can clearly see that the reduction of gridpoints is significant [Wae17].
For this thesis however the only important stencil is the direct neighbour stencil on a single color image. In Figure 2.4 the calculation of the interactions for such a stencil is illustrated.

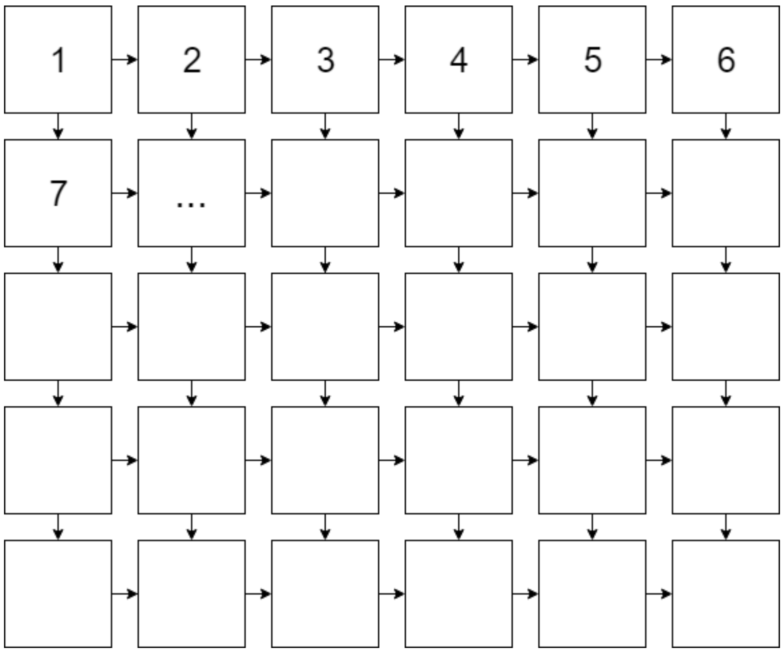


Figure 2.4: Direct neighbour stencil for a 6x5 image. An arrow represents a modeled interaction and the numbers represent the dimension of a pixel.

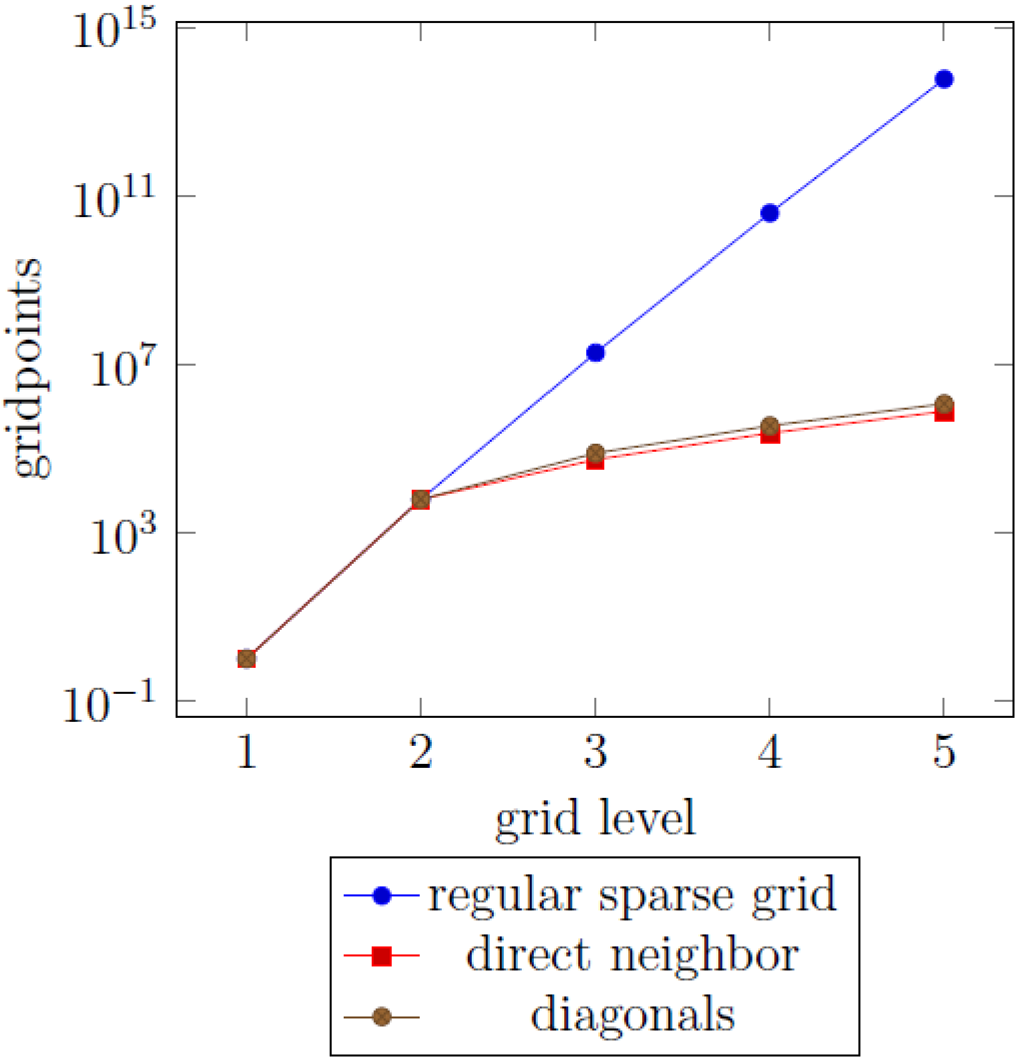


Figure 2.5: Amount of gridpoints for a 32x32 RGB image with different stencils compared to a regular SG. Graph from [Wae17]

3 Implementation

The implementation part of this thesis is mainly about a framework called SG++. It is a sparse grid implementation, in which a developer can make use of that library.
However, recently a data mining pipeline was added to make it also usable by non-developers. Up to this date the pipeline does not support geometry aware sparse grids, thus it is implemented in the span of this thesis.
SG++ also does not have many use cases to present the framework to people that are not familiar with it. Therefore a mobile application performing image classification using geometry aware sparse grids was developed.

3.1 SG++: General Sparse Grid Toolbox

SG++ is an open source numerical library for spatially adaptive sparse grids. It is written in C++ and was created by Dirk Pflüger throughout his dissertation in 2010. It is still being developed at the Chair of Scientific Computing at the Technical University of Munich.
One of the many features I will be working with is the datamining pipeline of the framework. The purpose of this pipeline is to make the use of the toolbox more user friendly. It uses a configuration file, which in our case is a JSON-file that stores the data that is supposed to be evaluated.
In this thesis the integration of the geometry aware sparse grids into the pipeline has been done. Now the user can choose the resolution of the image he wants to classify and specify the stencil in the configuration file.

3.2 Datamining Pipeline

The datamining pipeline in SG++ has the goal to make the toolbox easier to use. By using the pipeline for classification the only relevant components of the toolbox for the user is the *ClassificationMinerFromConfig* class [Fuc18]. The classification process can then be initiated by passing the configuration file as a command line argument to the execution of the *ClassificationMinerFromConfig* file.
The pipeline currently consists of four different components. However only the following two are relevant in this thesis

DataSource This module handles the data samples, that are used to train a model.

Fitter This module is the core of the pipeline as it implements various datamining methods.

3.3 Geometry Aware Sparse Grids Integration into Datamining Pipeline

For the datamining pipeline to support geometry aware sparse grids, a grid has to be created based on a specific stencil and the resolution of the image. With this stencil interactions get created and these interactions then get passed on to the grid creation. This created grid only contains gridpoints according to the interactions that have been passed on. The resulting grid is then used for training in the next steps of the pipeline.
Since the pipeline receives data through a configuration file, a *geometryConfig* has been added to the configuration file under the *Fitter* module with the parameters for the stencil and the resolution.
This new information for the geometry aware sparse grids is then processed in *ModelFittingDensityEstimationOnOff* by passing the parsed information in the *geometryConfig* onto the grid creation.
The grid creation used to be in the *ModelFittingBase* class, however for a better overview the grid creation is exported to the *GridFactory* class. *GridFactory* is now solely responsible for the grid creation in the pipeline.
Now it needs to be differentiated between creating a normal sparse grid and a geometry aware sparse grid. This is done by checking, if values in the configuration file for the *geometryConfig* have been passed on.
Now when it comes to creating the grid in the *GridFactory* class, a geometry aware

sparse grid is then created upon the stencil that has been chosen and the resulting interactions that get generated.
By this the geometry aware sparse grids have been integrated into the datamining pipeline.

3.4 Learning the data

The image classificator that has to be developed is supposed to classify hand drawn numbers. Therefore we need a large dataset containing hand drawn numbers to then train this data in the SG++ framework and finally export the trained data to evaluate it later on in the mobile application.

3.4.1 MNIST dataset

The dataset we chose for training our model is the *MNIST* dataset. This dataset is an accumulation of 60000 handwritten digitized numbers ranging from 0 to 9. Those images have a resolution of 28x28 pixels. As illustrated in Figure 3.1 the numbers have a white background and are written with a black color. The color white in the dataset has a value of zero and the color black has a value of one. It is also important to note that the pixel values of numbers almost always are values between black and white. This will be important later on for the mobile application [LCB].

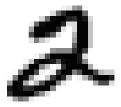


Figure 3.1: Sample image of the number two from the MNIST dataset
.

3.4.2 Training

The training was simple, because the functionality for the grid creation and density estimation is already implemented in the datamining pipeline. In Section 3.3 the geometry aware sparse grid support was implemented as well. The density estimation for the data is then done by using the *ClassificationMinerFromConfigFile* in the *sgpp/datadriven/examplesPipeline* and adding the created *geometryAwareSparseGrid.json* configuration file as a command line argument to it.
The *geometryAwareSparseGrid.json* contains two specific values for geometry aware sparse grids:

- On the one hand a resolution parameter, which in this case has to be 28x28 matching the provided dataset in Section 3.4.1.
- And on the other hand a stencil parameter, which we chose to be the *DirectNeighbour* stencil.

3.4.3 Exporting

The data is simply exported by storing labels, instances, grids and surpluses of each model into text files. This process was split up by having one text file for all labels, one text file for all instances and ten text files for each grid and surpluses of a model.
The data can be accessed through the *ModelFittingClassification* class, because it stores all the references of the current models. Additionally it stores the labels and instances of the current models. So by extending the *ModelFittingClassification* class by a *storeClassificator()* method, we can already save labels and instances into text files.
A model is created in *ModelFittingBaseSingleGrid* and contains the created grid and calculated surpluses of a *Classificator*. Therefore *ModelFittingBaseSingleGrid* has to be extended by a *storeFitter()* method that converts and returns the grid and surpluses together as strings.
Now by calling *storeFitter()* for every model in *storeClassificator()* we also receive the grids and surpluses.
With this we have all data needed to classify new unseen data and can move on to develop our mobile application and implement an evaluation method based on the exported data we have.

3.5 App development

One goal of this thesis was to develop a mobile application that demonstrates image classification through geometry aware sparse grids. This was supposed to be working on a standalone tablet, however executing the calculations solely on the tablet was not feasible and I developed a workaround for this, which includes a server.
To start off with the development the exported data has to be imported into the server and structured so it can be easily accessed for the evaluation.
Then for the *Server-Client Communication* that is illustrated in Figure 3.2 a frontend application, which will be called *Client* further on and a backend application, which will be called *Server* further on has to be developed.

HUAWEI Mediapad M5 will be the *Client* throughout this thesis. It is a tablet with an Android operating system with a resolution of 2560x1600 pixels. The information of the resolution will be of big importance, as we will see during the developing process of the application.

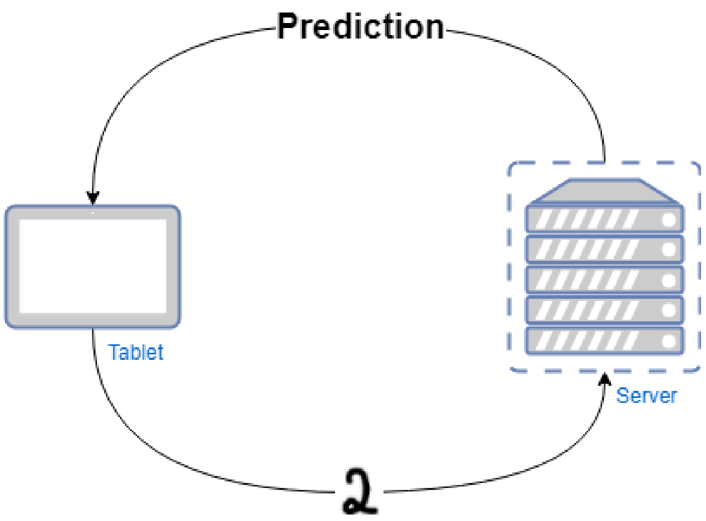


Figure 3.2: Simplified Server-Client communication. The procedure of the prediction is to draw a number onto the touchscreen of the tablet and press on predict. Then a message is sent to the server, which then evaluates the number and returns back a prediction of the number. The prediction is then displayed on the tablet.

3.5.1 Data Structure

As we know already we exported text files containing the necessary information. To perform an evaluation on this data, we need to create a data structure in Java. As seen in Figure 3.3, we split the data into three classes.

Classificator An instance of the classificator class represents a trained model, in our image classification of handwritten numbers each of those instances represent a number ranging from 0 to 9.

Grid The grid class stores two lists. The first list *surpluses* are the exported weights of the density functions stored as *Double*.
The second list *gridPoints* is a double list. Whereas the size of this list represents the amount of grid points, which in our example are 10753 points for each model. The list

inside *gridPoints* represents the position of each grid point of that specific grid, which in our example is a position in 784 dimensions.

GridPoint The *GridPoint* class stores the level and index of a specific grid point inside a grid.

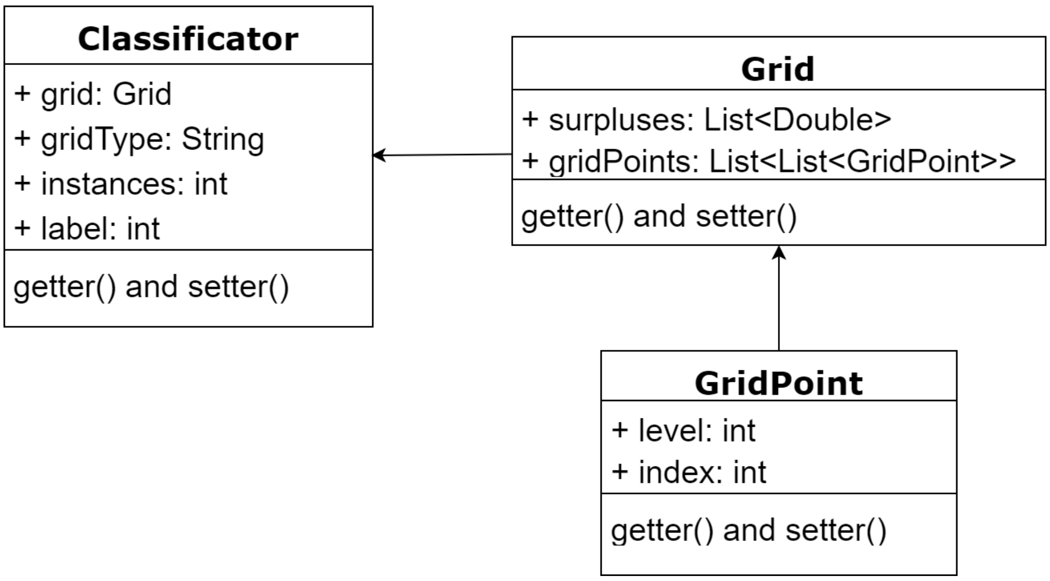


Figure 3.3: Data Structure in Java of the exported state

3.5.2 Client

The client, as illustrated in Figure 3.2, is a tablet with an Android operating system. Therefore the application for the image classification was developed in Android Studios. The layout is written in XML and Java classes handle the functionality of those XML-objects that are displayed. In Figure 3.8 we can see a screenshot of the application. In the following I will briefly discuss every feature of the client and explain some decisions that had to be made [Goo].

Color of drawing and background As already explained in Section 3.4.1 the numbers of the dataset are drawn with a black color on a white background and to match the dataset we will also have black draw color and white background.

Drawing Window The goal was to have a drawing space inside the app, that creates an image with the same resolution as the training data set. The required resolution is an image of 28x28 pixels, however the resolution of the HUAWEI Mediapad M5 tablet is 1600x2500 pixels. First off all, to avoid bending or stretching of the image, we will create a canvas that has the same width and height. Now we have to bring the image that is drawn into the same size as the training dataset of 28x28 pixels. There are several approaches to achieve the expected outcome.

- One Solution could be to consecutively bundle 4 pixels until we reach the ends of the tablet. This iteration would end after two steps making a final resolution of 448x448 pixels. Then one would just have to add every 16th pixel into a list and return it

- Another solution could be to make use of the most space. By defining the drawing space to 1600x1600 pixels and downscale the image afterwards to the required 28x28 pixels and returning this image

The downsides of the first solution are obvious, since the drawing space would be relatively small compared to the actual size of the tablet. 448x448 pixels is the biggest resolution that can be achieved by this approach, due to to next being 1792x1792 pixels. The tablet however only has a width of 1600 pixels. Also every pixel in the returned list would be either black or white, this does not match the training data set. The smartest way to attain the best prediction results would be to have an input that also matches the training data. The data contains various values ranging from black to white, therefore this approach is not a good fit.
The second solution seems more promising than the first one. First off we have the biggest possible square drawing window with 1600x1600 pixels, this makes the user experience way better. Additionally by downscaling the image with the integrated function of Android Studio *Bitmap.createScaledBitmap()* we get interpolated values of the drawn pixels. That means the image we want to predict matches the training dataset even more, because it mostly consists of interpolated values between 0 and 1.

PREDICT Button The button itself is self explaining. After drawing an image into the *Drawing Window*, the user can press the *PREDICT Button* to have his drawing evaluated.

CLEAR Button If this button is pressed the entire *Drawing Window* will be cleared of any current drawing. This is done by calling the method *clearBitmap()* onto the drawing canvas. Additionally, if a *PREDICTION* is currently displayed it will be made invisible.

SETTINGS Button This button redirects to another activity, in which settings can be adjusted. There are two settings that can be adjusted currently

- Since the application only supports local routing up until now. The IPv4 adress has to be adjusted, if the server is connected to another local network. Therefore the user can change the IPv4 adress easily in the *SETTINGS* activity.
- For developers datasets can be created with the *STORE* and *SAVE* functions. To recognize them later a filename can be set for the current dataset

Display Prediction After having pressed on the button *PREDICT*, a message is displayed in the center top of the screen. This message lasts there until the *CLEAR* button is pressed.

STORE Button This button is a button for developers, thus it is not shown in the screenshot. This button triggers the method *store()* in the *MainActivity*. By pressing it a String is appended with the current bitmap that is displayed.

SAVE Button This button is a button for developers, thus it is not shown in the screenshot. This button triggers the method *save()* in the *MainActivity*. By pressing this button every image that was stored by pressing *STORE* is saved in a file in the *INTERNAL STORAGE* of the device and can then be used as dataset for further evaluation.

Unused space By having a square *Drawing Window*, one third of the screen is left unused. Therefore to make the user interface more appealing an explanation of the classification method is given. Also the currently supported numbers that can be predicted are displayed. Lastly the TUM logo is added to top off the appearance of the interface.

This was a small explanation of every feature of the user interface. Now I want to move on and explain two of the most important functions of the client.

getPoint() This method is essentially the formatting of the number that is being drawn into the desired state that we talked about earlier. To format the data we basically have to do three things:

- Scale down the original image of the tablet into to required resolution of 28x28 pixels. This is done consecutively by using the *Bitmap.createSclaedBitmap()* method of the Android Studio library. With this we basically scale the image step by step to get interpolated values. These values match better with the training data. If the downscale would have been done in one step, almost every value would either be black or white. This different downscaling approaches can be seen in Figure 3.4.

Also comparing Figure 3.6 and Figure 3.5, it can be clearly seen that Figure 3.6 matches the *MNIST* dataset more.

- Arrange the pixel values accordingly to the MNIST dataset. This is also done to match the dataset, because the training dataset is iterated over row wise and not column wise like a usual list would be.
- Normalize the values to match the training data, due to color values in Android Studio differing from the color values of the MNIST dataset.

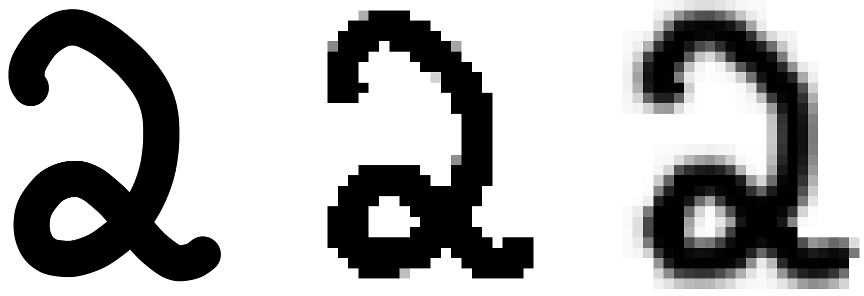


Figure 3.4: Original image(left), one time down scale(center) and consecutive down scale(right)

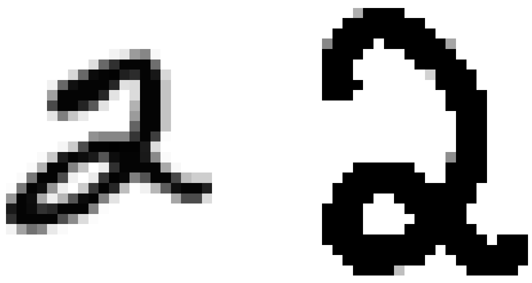


Figure 3.5: Sample from MNIST Dataset(left) and sample of a one time downscale(right)

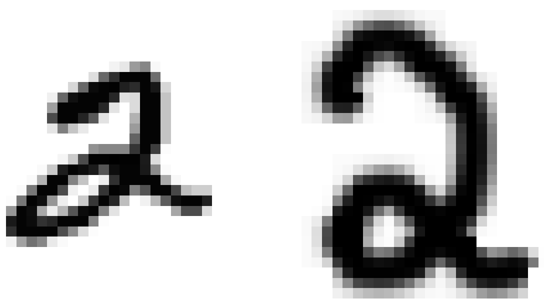


Figure 3.6: Sample from MNIST Dataset(left) and sample of a consecutive downscale(right)

predict() This method is the *onClick()* method of the *PREDICT* button. The procedure of this method is illustrated in Figure 3.7.

1. The *getPoint()* method is called to format the data into the desired state.
2. The data is then formatted to a JSON-object.
3. Either previous IPv4 address is used or updated address from *SETTINGS* is used.
4. Send HTTP POST Request to *http://Host:Port/predict*.
5. Wait specific time for response.
6. If response is received the prediction is displayed, otherwise error message can be seen from console.

3.5.3 Server

The servers responsibility is to standby until a message is received, which contains the data of the number that was drawn on the tablet. Then to evaluate this message, classify the number and lastly return a message of the predicted number in form of a JSON-object.
The server is setup on a Spring Boot framework and written in Java. Therefore it can be run on any machine that supports a version of Java 1.8 or higher. Preferably a more powerful machine, since reading the trained data and creating objects according to our predefined data structure in Section 3.5.1 requires a lot of RAM and some computation time depending on the processor used.

Spring Boot Spring Boot is an extension to the Spring framework. It allows us to easily

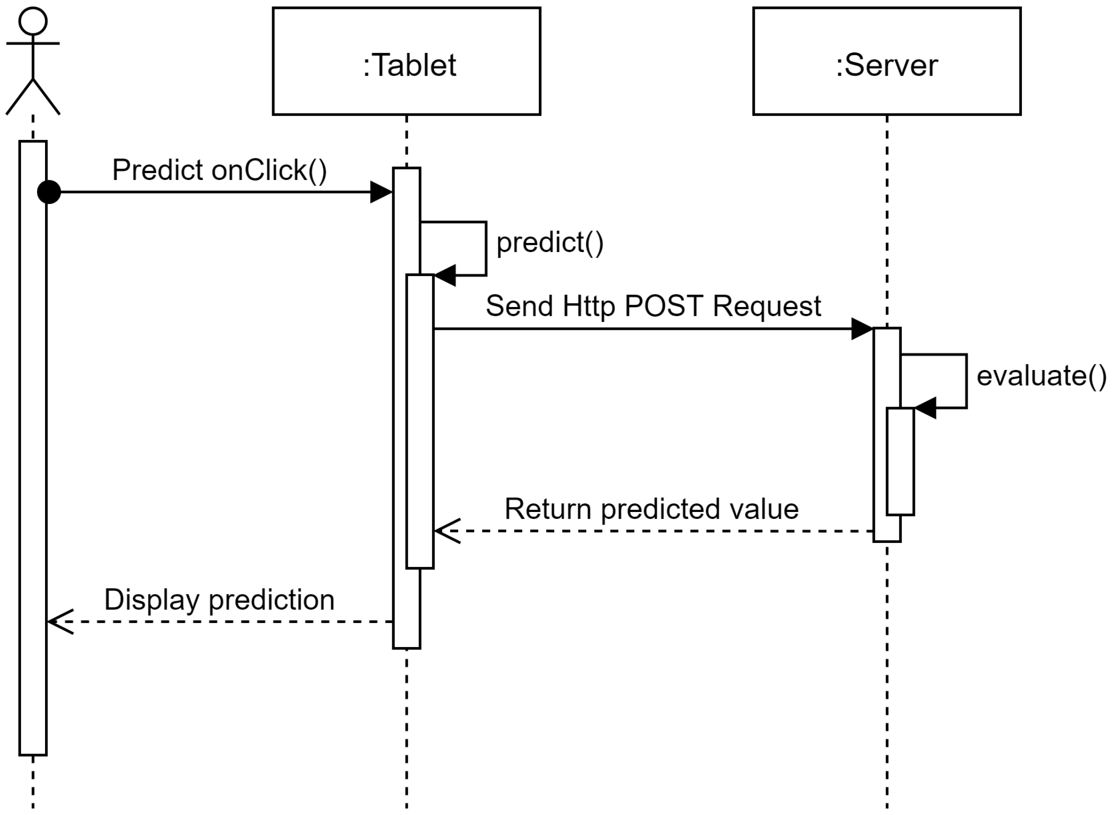
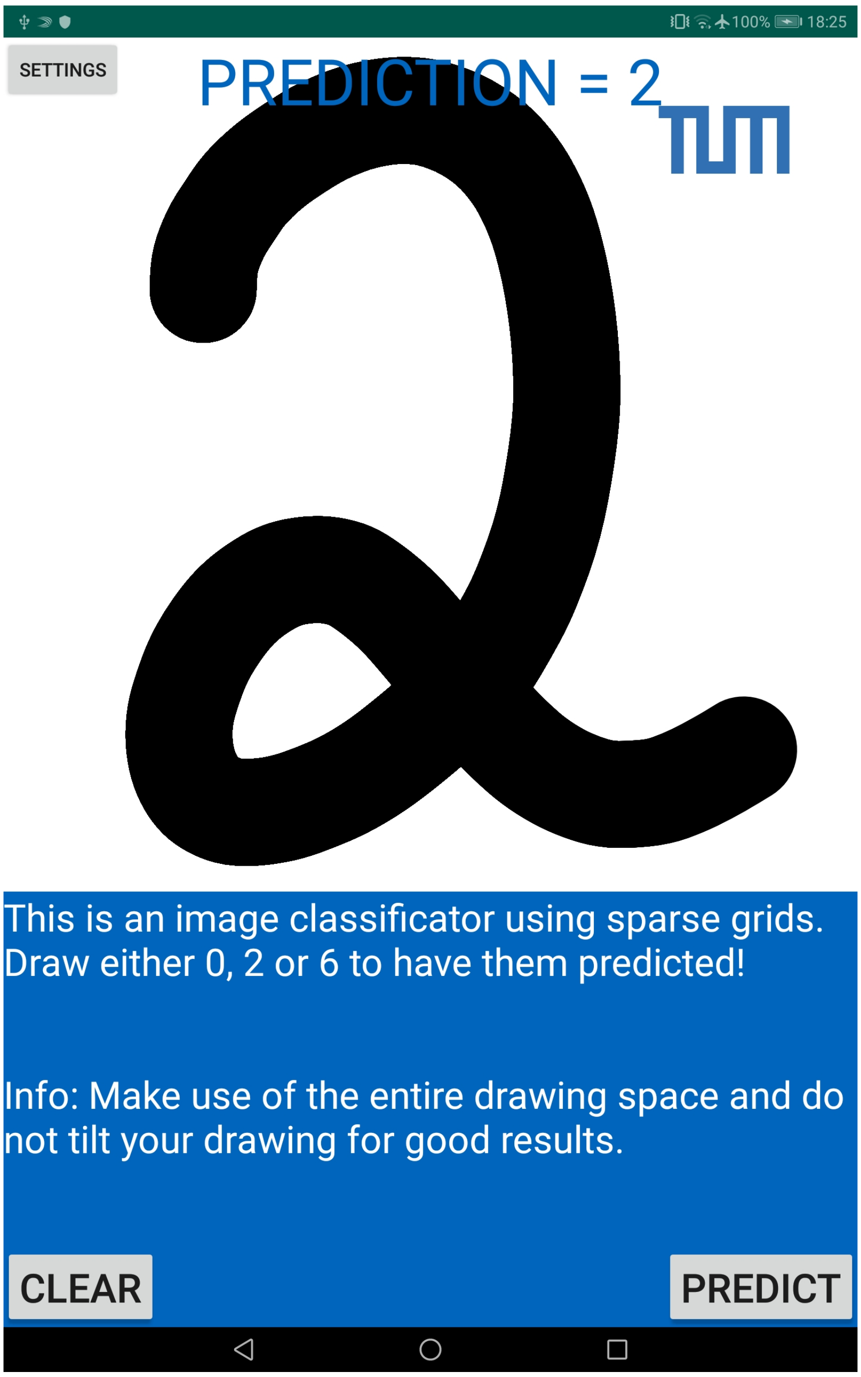


Figure 3.7: Sequence Diagram of Server-Client Communication



This is an image classificator using sparse grids. Draw either 0, 2 or 6 to have them predicted!

Info: Make use of the entire drawing space and do not tilt your drawing for good results.

Figure 3.8: Screenshot of the SGDE App, taken from the HUAWEI Mediapad M5

setup a server, without having to make further configurations. By creating a Spring Boot project with a REST API, a server is hosted on *http://localhost:8080*. Localhost is the IPv4 address of the machine the server is running on, we can configure our tablet to send a message to this address under *SETTINGS* [Sof].

Handle HTTP POST Request As already mentioned earlier we use a HTTP POST Request in the Client to have the number predicted. On the server side POST allows us to receive some type of data as JSON-object, do something with the data and finally return some type of data as JSON-object. In our context we receive the image as JSON-object, convert the data of the image, perform an evaluation and finally return the label of the predicted number. The POST request can be made through *http://localhost:8080/predict*.

Data Structure We construct our data structure for the *Classificators* according to Figure 3.3.

Beans This project was supposed to be solely run on the tablet. However converting the exported data from Section 3.4.1 into our preferred data structure from Section 3.5.1 has a high computational effort, due to this data structure creating many objects. Therefore the application on the tablet would either load for a long time every time it is opened or it could also simply run out of memory. Therefore a workaround had to be developed to have a good user experience.
This problem is solved by using the *Bean* concept of the Spring framework. It allows us to convert the data into the data structure at the start up of the server. That means we use the computational power of the server instead of the tablet to fasten the process of parsing. Due to the server running in the background, the user can open, close and run the application without delay.
In this example we define ten *Beans* representing each *Classificator* and parse it to the desired data structure in *ClassificatorFactory*. Now we can have access to this data at run time.

With this brief overview of the server we can go over to the actual implementation of the *evaluation()* method.

evalClassificator() is the first part of the implementation for the *evaluation()* method and can be seen in Algorithm 1.
As explained previously in the *Beans* and *HTTP POST Request* sections, we start off by having all *Classificators* loaded and are already in a state to handle an incoming *POST Request*. If that *POST Request* is valid *evalClassificator()* is called with the set of

Classificators C and the point *P*. *P* is a set of pixels that represent the image. Also in this context the validation of the *POST Request* is done by verifying the size of *P* being 784 pixels.
Moving on to the algorithm, first the total amount of instances the data was trained with is calculated. This value might or might not be used, depending if prior is set or not. Then for every *Classificator* the density values are calculated. This is done by calling the *evalSparseGrid()* method illustrated in Algorithm 2.
After the *classDensity* calculation, the algorithm checks for the prior value. If it is set a relative density is calculated, if it isn't then the *classDensity* values are unchanged. The effect of prior can be seen in Table 4.3 and Table 4.9.
Finally the label of the largest *classDensity* value of all densities is returned.

evalSparseGrid() is called in *evalClassificator()*. It is invoked with a point and a specific *Classificator* that the point should be evaluated on. This method returns the actual density value.
Since the *Classificator* already provides every necessary information that we need to evaluate the point, the algorithm just extracts and initializes every information needed. Additionally to that, a *sum* variable is initialized, which will be our return value in the end. A grid of a *Classificator* contains grid points. Each grid point has positional information in form of a level and index, in every dimension.
Now the algorithm just iterates over every grid point and for each position in every dimension the point in located in, $\varphi_{l,i}(\text{P[dimension]})$ is calculated.
$\varphi_{l,i}(\text{P[dimension]})$ is in this case the modified linear basis function evaluated on the image.
$\varphi_{l,i}(\text{P[dimension]})$ is then multiplied for each iteration.
After this is done for one point the *sum* variable is updated by adding the weighted basis function to it.
Finally the *sum* variable is returned, which is our required *classDensity*.

Algorithm 1: evalClassificator()

```
Data: set of classificators C, point to be evaluated P, prior
Result: label of predicted value
sumInstances $\leftarrow$ 0;
for each classificator $c \in C$ do
    sumInstances += c.getInstances();
end
for each classificator $c \in C$ do
    classDensity $\leftarrow$ evalSpraseGrid(c, P);
    if prior then
        relativeAmount $\leftarrow \frac{\text{c.getInstances()}}{\text{sumInstances}}$;
        classDensity $\leftarrow$ classDensity $\times$ relativeAmount;
    end
    classDensityList.add(classDensity, c.getLabel());
end
return maxClassDensity(classDensityList).getLabel()
```

Algorithm 2: evalSparseGrid()

```
Data: classificator c, point to be evaluated P
Result: classDensity
sum ← 0 ;
product ← 0 ;
currentIteration ← 0;
dimension D ← c.getDimensions();
grid = c.getGrid();
surpluses S = grid.getSurpluses();
gridPoints GP = grid.getGridPoints();
for each gridpoint gp ∈ GP do
    for each dimension do
        if dimension == 0 then
            product ← $\varphi_{l,i}$(P[dimension]);
        else
            product ← product × $\varphi_{l,i}$(P[dimension]);
        end
    end
    sum += product × S[currentIteration];
    currentIteration++;
end
return sum;
```

4 Evaluation

In this chapter our sparse grid implementation will be validated by comparing the outcome with the SG++ sparse grid implementation on different datasets. Also the accuracy of our sparse grid implementation will be examined on different datasets.

4.1 Validation of Sparse Grid Implementation

First of we want to validate our sparse grid evaluation implementation that has been done in Section 3.5.3. We can do this by computing the relative error of the evaluation function of our implementation and comparing it to the evaluation function of the SG++ framework. This is done on a density function that was computed in SG++ from the *MNIST* dataset for the number 2.

Let x be our expected value and x_0 be our measured value. Then we get the absolute error by calculating the difference Δx and taking the absolute value of it. Then by dividing the absolute error by the absolute expected value we get the relative error δx.

$$\delta x = \frac{|\Delta x|}{|x|} = \left|\frac{x_0 - x}{x}\right| = \left|\frac{x_0}{x} - 1\right| \tag{4.1}$$

Now we can convert the relative error into a percentage error $pe(x_0, x)$.

$$pe(x_0, x) = \left|\frac{x_0}{x} - 1\right| \times 100\% \tag{4.2}$$

To get a more accurate percentage error, we can calculate it for many values and then take the mean of it. Let *listOne* be of size N containing all measured values and *listTwo* also be of size N containing all expected values. Then we get the mean percentage error $mpe(listOne, listTwo)$ with

$$mpe(listOne, listTwo) = \frac{\sum_0^n pe(listOne.get(n), listTwo.get(n))}{N} \tag{4.3}$$

Applying (4.3) to our case we view the *evaluation()* from the framework as expected value and the *evaluation()* from the server as measured value. In the following the mean percentage error is calculated on two different datasets.

4.1.1 Random dataset

The first dataset we want to validate our sparse grid implementation on is a randomized dataset. For this a dataset with 1000 points was created. Each datapoint has a list of the size 784, which represents a 28x28 pixel image. The values inside an image are calculated by the randomize function of python $random.uniform(lowerBound, upperBound)$. For our case we need a $lowerBound = 0$ and an $upperBound = 1$. Samples of this data set can be seen in Figure 4.1.

Calculating the mean percentage error is done by evaluating both datasets on the density estimation for the number two. We get

$$mpe(evaluationServer(randomDataset), evaluationSG++(randomDataset)) = 313.19460522364295\% \approx 313\% \quad (4.4)$$

As we can see the mean percentage error of this dataset is very high with approximately 313%, this might mean that the algorithm is not correctly implemented.

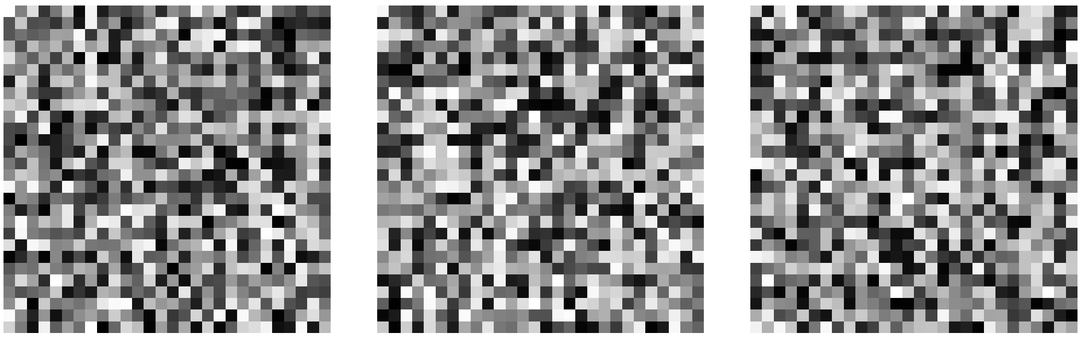


Figure 4.1: Examples of randomized dataset

	Density values	
Point	SG++	Server
1	-161.27	-157.94
2	128.69	255.09
3	111.24	81.04
4	268.05	452.29
5	41.76	78.56
6	-111.23	-297.62
7	-16.34	-49.09
8	-326.19	-384.61
9	316.23	337.64
10	141.27	159.13

Table 4.1: Some sample density values from the evaluation of the random dataset on the *Classificator* 2 in the server and in SG++

4.1.2 Tablet dataset

The next dataset is generated through the tablets user interface. Since the data is evaluated onto the density estimation of the class label two, it makes sense to also draw the number two and create a dataset of it. The images drawn are already in the right format. Examples of this dataset can be seen in Figure 4.2
So by just exporting them and evaluating this dataset onto both *evaluation()* functions we get a mean percentage error of

$$\begin{aligned} &mpe(evaluationServer(tabletDataset), evaluationSG++(tabletDataset)) \\ &\qquad = 0.7542007256638145\% \approx 0.75\% \end{aligned} \tag{4.5}$$

This value is very very small compared to (4.4). The measured values of this dataset barely differ from the expected values.

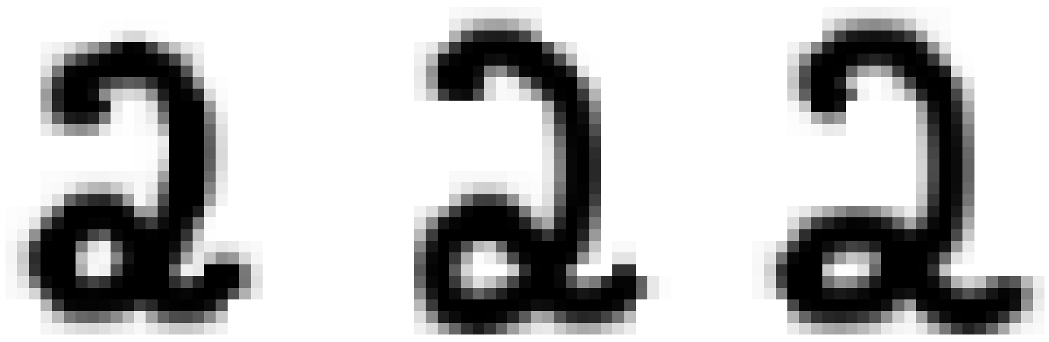


Figure 4.2: Examples of tablet dataset

	Density values	
Point	SG++	Server
1	25717.54	26008.06
2	24826.65	24953.14
3	23539.39	23649.79
4	24847.33	24996.17
5	23095.78	23289.79
6	23633.21	23819.00
7	25665.37	25853.70
8	22972.48	23194.50

Table 4.2: Some sample density values from the evaluation of the tablet dataset on the *Classificator* 2 in the server and in SG++

4.1.3 Outcome

By looking at the values of (4.5) and (4.4) and the context of the use we can answer this question.
The mean percentage error in (4.4) makes it very clear that there is something not perfect in our sparse grid implementation. Meaning that either exporting the learned state has flaws and/or the evaluation function in the server is calculated wrong.
However the mean percentage error in (4.5) proves the opposite. If actual valuable images are provided the sparse grid implementation on the server has a very low relative error. This means that the evaluation function has to be correct, but exporting the state was not done perfectly and a rounding error is made somewhere.
The only values that might cause this rounding error have to be the surpluses that are

exported in Section 3.4.3. Since every other value that is exported is of the type Integer or String.
Also the other thing we can observe are the actual values of the densities that can be seen in Table 4.1 and Table 4.2. The random dataset densities are usually around the number 0, whereas the densities for valuable data from the tablet is around 24000. This pattern for the densities can be observed for every valuable image. A valuable image is a hand drawn number between 0 and 9. That means that the relative error is also very little for any other number.
All in all this implementation of the evaluation function of a sparse grid density estimation that is done here is still valid. The use of this sparse grid implementation is to predict numbers, so only valuable information counts. In Section 4.1.2 it is proven that the prediction of valuable pictures have a negligible relative error.

4.2 Accuracy of sparse grid implementation

After validating our sparse grid implementation, we can move on to actually testing and evaluating the classification result we get from our evaluation function on the server. We will do this by using different datasets. Including the MNIST test dataset and datasets created by drawing onto the tablet.

4.2.1 Testing process

Here I want to explain the process and the conditions I tested my sparse grid implementation on.

Choice of numbers Basically the numbers I will be testing on are 0, 2 and 6. This is due to dominant number problem. This problem is already known and discussed in [Wae17]. To give a small explanation, in the trained data there exist some numbers that are more dominant than other numbers. A dominant number tends to get predicted more often than others. In [Wae17] it is reasoned by numbers that have the least variation of pixels result in being more dominant. The pattern of the dominant numbers according to [Wae17] are as following: 1, 9, 7, 4, 6, 8, 3, 5, 2, 0 with 1 being the most dominant and 0 being the least dominant.
The result of the evaluation of all numbers 0 to 9 from the MNIST test dataset on our application can be seen in Table 4.9. The results are very poor and support the dominant numbers problem since 1 is always predicted. 1 being always predicted matches the pattern from before. This means that it makes no sense evaluating the application on all numbers 0 to 9 that the data provides. Instead the numbers 0, 2 and 6 are used, since according to [Wae17] 0 and 2 are numbers that have similar standard

deviation therefore will have good results. The 6 was also added to have more variety in the predicting process. This makes the application more fun to the user. Also the 6 is a good number to use since it differs a lot from 2 and 0 pixelwise and therefore I am making the assumption that this will lead to better results.

Automation of process For this a .python file was written that parses the datasets, sends *HTTP POST Requests* to our server and then evaluates the result.

4.2.2 MNIST datasets

The first dataset we are testing our sparse grid implementation on is the *MNIST Test Dataset*. Since we want to get the accuracy on the actual user application we are only testing it on 0, 2 and 6. Thus we only get those three datasets out of the test data.

MNIST Test Dataset			
Number	Number of Images	Prior	Accuracy
0	980	No	0.7653
0	980	Yes	0.7724
2	1032	No	0.7161
2	1032	Yes	0.9099
6	958	No	0.9937
6	958	Yes	0.9833

Table 4.3: MNIST Test Dataset for numbers 0, 2 and 6 evaluated on our sparse grid implementation

Prior The dataset was tested with and without setting prior. As a reminder, if prior is set the amount of instances, that the densities are trained on, have an impact for the evaluation. In this example we can see the accuracy for 0 increases a bit and the accuracy for 2 increases a lot, while the accuracy for 6 decreases a little as well. Based on these results we will have prior set for the user application to get better results. Since the accuracy for the number 2 increases by 20%, while the accuracies for 0 and 6 are barely affected.

Accuracy The accuracy of these results are very promising. If a 6 is drawn it almost always gets predicted with a 99% chance, which proves my assumption from earlier that the number 6 differs a lot from 0 and 2 and thus results in better prediction for the number 6. With prior being set number 2 has a good result with around 90% accuracy and number 0 has an acceptable result with around 77% accuracy.

4.2.3 Handdrawn datasets

The next datasets we are going to be using are datasets created by people drawing onto the tablet.

User 1 The results of user 1 can be seen in Table 4.4. For numbers 2 and 6 the accuracy of the prediction is really good, however the accuracy for the number 0 is very poor. This poor result can be reasoned by the user not taking advantage of the full size of the drawing space. This can be seen in Figure 4.3. All samples of the number 0 are quite small and have similarities to the circle that a 6 contains.

User 1			
Number	Number of Images	Prior	Accuracy
0	26	Yes	0.2308
2	27	Yes	0.9259
6	30	Yes	0.8

Table 4.4: Dataset of user 1 evaluated for numbers 0, 2 and 6 on our sparse grid implementation

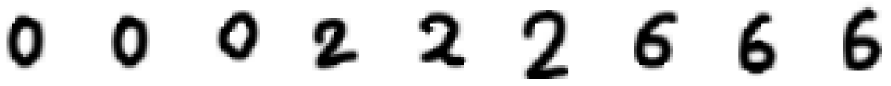


Figure 4.3: Samples from user 1

User 2 The results for user 2 can be seen in Table 4.5. Overall the accuracy is very good. The samples from user 2 in Figure 4.4 make use of the full screen and are not tilted significantly, this results in a good accuracy.

User 2			
Number	Number of Images	Prior	Accuracy
0	23	Yes	0.8261
2	13	Yes	1.0
6	36	Yes	0.9162

Table 4.5: Dataset of user 2 evaluated for numbers 0, 2 and 6 on our sparse grid implementation

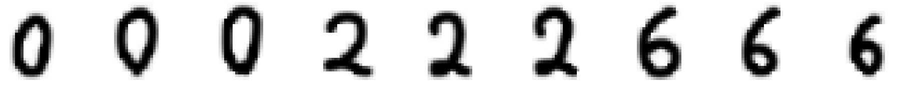


Figure 4.4: Samples from user 2

User 3 The same thing applies here that applied to user 2. The numbers are not tilted significantly and use the entire screen. This explains the good accuracy in Table 4.6.

User 3			
Number	Number of Images	Prior	Accuracy
0	13	Yes	0.8462
2	17	Yes	1.0
6	14	Yes	0.9286

Table 4.6: Dataset of user 3 evaluated for numbers 0, 2 and 6 on our sparse grid implementation

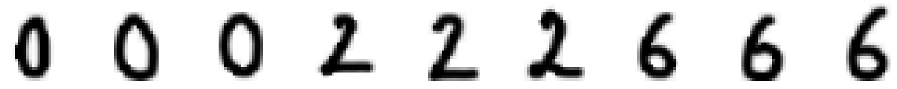


Figure 4.5: Samples from user 3

User 4 A very low accuracy can be observed in Table 4.7 for the number 2. Looking at Figure 4.6 the samples for number 2 are drawn in a tilted way. This makes it hard to predict the number, since it is not trained that way. However numbers 0 and 6 are predicted right with a very good accuracy.

User 4			
Number	Number of Images	Prior	Accuracy
0	51	Yes	0.9804
2	20	Yes	0.05
6	20	Yes	1.0

Table 4.7: Dataset of user 4 evaluated for numbers 0, 2 and 6 on our sparse grid implementation

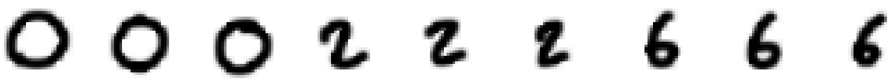


Figure 4.6: Samples from user 4

User 5 Table 4.8 shows poor results for the numbers 0 and 2. The samples for 0 and 2 in Figure 4.7 have the same properties as the previous numbers that have been predicted poorly. They are small and tilted. In this case the 6 has a relatively low accuracy compared to the usual accuracy of the number 6. Here the 6 is also drawn quite small.

User 5			
Number	Number of Images	Prior	Accuracy
0	26	Yes	0.2353
2	27	Yes	0.2778
6	30	Yes	0.7647

Table 4.8: Dataset of user 5 evaluated for numbers 0, 2 and 6 on our sparse grid implementation

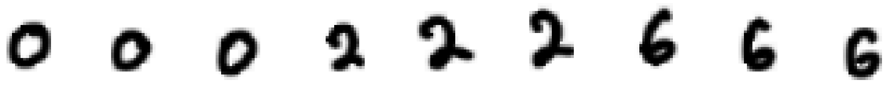


Figure 4.7: Samples from user 5

4.2.4 Outcome

Generally the accuracy for both the test dataset from MNIST and user datasets created on the tablet have promising results.
However for users that tend to draw small, the accuracy decreases a lot. This phenomena can be seen in the results for users 1, 4 and 5. This could be fixed by downscaling the drawn image in a different way. For example by extracting the extra white space, if a user draws a small image.
Also there seems to be a decrease in the accuracy, if the number is drawn in a tilted way. This can be seen in the results of user 4 and 5.
Concluding for getting the best results for the current implementation it is best to use the entire screen to draw on and to not tilt the drawing. By following these conditions an average of 90% accuracy can be guaranteed.

MNIST Test Dataset			
Number	Number of Images	Prior	Accuracy
0	100	No	0.29
0	100	Yes	0.0
1	100	No	1.0
1	100	Yes	1.0
2	100	No	0.03
2	100	Yes	0.0
3	100	No	0.06
3	100	Yes	0.0
4	100	No	0.36
4	100	Yes	0.0
5	100	No	0.02
5	100	Yes	0.0
6	100	No	0.26
6	100	Yes	0.0
7	100	No	0.37
7	100	Yes	0.0
8	100	No	0.01
8	100	Yes	0.0
9	100	No	0.29
9	100	Yes	0.0

Table 4.9: MNIST Test Dataset for all numbers evaluated on our sparse grid implementation

5 Conclusion

Over the span of this thesis geometry aware sparse grids have been integrated into the datamining pipeline of SG++. This required adding specific geometry aware parameters to the configuration file that is used to instantiate the pipeline. Those parameters are then parsed and used during the grid creation.
Additionally a mobile application to demonstrate image classification with geometry aware sparse grids was developed in this thesis. The application is a hand drawn number classificator. For this the previously added support for the geometry aware sparse grid in the pipeline was utilized. The pipeline was used to create the grid and the density functions. This data then had to be exported and evaluated on a embedded sparse grid implementation. The embedded implementation was constructed by a Server-Client application, since the Client alone was not able to handle processing the data in a reasonable time.
During evaluation a few adjustments had to be made on the application. First off the embedded sparse grid implementation had a very high deviation for the evaluation function compared to the SG++ evaluation function. However this high deviation was only apparent for random datasets and the deviation was very low for so called *valuable* datasets. Valuable datasets are basically datasets that represent numbers. Since numbers are the only data that are evaluated in the actual user application, the implementation could still be used further on. Secondly there was a high inaccuracy, if all numbers were taken into account in the classifying process. This was reasoned by the *dominant number problem*. To avoid this problem numbers were chosen that have similar standard deviation and/or numbers that are very different pixelwise.
Overall the first prototype of the application shows promising accuracy for the chosen numbers. For the large *MNIST Test Dataset* an average accuracy of 88% for the prediction of a number can be promised. However for users that tested the application, various results were observed. Some users even exceeded the *MNIST Test Dataset* accuracy, whereas some were far below that accuracy. This was due to some user having very different handwriting, but mainly was caused by tilting their drawing or drawing very small numbers into the tablet.

6 Future Work

Work that still can be done starting off from this thesis can be partitioned in two different areas.
First off the datamining pipeline currently only supports the *Direct Neighbour Stencil*. The stencils can easily be expanded with all the stencils mentioned in [Wae17]. Including the *Diagional Stencil* and the *Square Stencil*. Also refinement for geometry aware sparse grids in the pipeline has to be added.
In this thesis exporting the trained data from the pipeline was done ad hoc, this process however can be implemented into the pipeline as extra feature that the user can select in the configuration file.
The second part is the developed user application. With the current user application merely just being a first prototype there is lots of room to improve.
As for the frontend the only supported layout is vertically and with a resolution of 2560x1600 pixels. This is due to the image classification requiring specific input data, therefore for simplicity the layout was designed with the tablet that was provided. This means as for now the application was not tested on different devices with different screens and will be most likely displayed incorrectly.
Also a horizontal layout for the current device is not supported and can also be easily extended.
There are lots of parameters used for the drawing, these parameters can be changed and can then be examined for the accuracy of the prediction. Different pencil sizes or different drawing patterns might lead to better results, this can be tested and documented.
Also in the current implementation the down size of the picture is done by the library function of Android Studios. Different down sizing methods can be applied and tested for accuracy as well.
Also the user application can be extended to support different modes of classification. For example the user could be able to choose the datasets, i.e. different numbers or shapes, he wants to classify or choosing a mode like *SUMMATION*, in which the user can draw two numbers that get summed up.
The application can also be extended to support error handling. For example a blank drawing will not throw an error in the current application.
For the backend we found out in Section 4.1 that there is a big deviation for random

datasets in the embedded sparse grid implementation compared to the SG++ implementation, this deviation has to be investigated for clarification. However does not make a big difference for the current application.
Also the backend needs trained data, that is created through the tablet for better classification results. For now the trained data only consists of data from *MNIST*. This can easily be done, since the tablet already supports dataset creation, however it would be very tedious for one person to create a big enough dataset. Therefore the process of data creation has to be automated and collected from users of the application.
The current application only works for local networks. Meaning that the server and client have to be in the same network to communicate. This can be made globally accessible through the web by creating a server that runs constantly and making the current localhost IPv4 address accessible from the internet.
To support different datasets the backend has to be extended with different *POSTMapping* methods that include different datasets.

List of Figures

List of Tables

Bibliography

[Fuc18] D. Fuchsgruber. “Integration of SGDE-based Classification into the SG++ Datamining Pipeline.” Bachelor Thesis. Technical University of Munich, Aug. 2018.

[Goo] Google. *Android public API documentation*. Accessed: 15.12.2018. URL: `https://developer.android.com/docs`.

[LCB] Y. LeCun, C. Cortes, and C. J. Burges. *THE MNIST DATABASE of handwritten digits*. Accessed: 20.01.2019. URL: `http://yann.lecun.com/exdb/mnist/`.

[Peh13] B. Peherstorfer. “Model Order Reduction of Parametrized Systems with Sparse Grid Learning Techniques.” Dissertation. Technical University of Munich, Aug. 2013.

[Pfl10] D. Pflueger. “Spatially Adaptive Sparse Grids for High-Dimensional Problems.” Dissertation. Technical University of Munich, Feb. 2010.

[Sof] P. Software. *Spring Boot*. Accessed: 10.02.2019. URL: `https://spring.io/projects/spring-boot#overview`.

[Wae17] T. Waegemans. “Image Classification with Geometrically Aware Sparse Grids.” Bachelor Thesis. Technical University of Munich, Oct. 2017.