Visualization of the Varying Spatial Density Information in the Swiss Feed Database

Bachelor Thesis in Computer Science

provided from:

Andrin Betschart

Zofingen, Switzerland

Made at the

Department of Informatics

at the University of Zurich

Prof. Dr. M. Böhlen

Supervisor: Andrej Taliun

Delivery date: August 16th, 2012

Acknowledgments

I would like to express my deep gratitude to Dr. Andrej Taliun, my supervisor who supported me through the entire work. His patient guidance, enthusiastic encouragement, explanations and useful critiques contributed in a remarkable way to the success of this thesis. My grateful thanks also goes to Prof. Dr. Michael Böhlen, who gave me the chance to work on a project, which was interesting and challenging.

Special thanks also to Annelies Bracher of the Swiss Agronomic Institute for supporting me during the evaluation and providing me with useful information from the users' side.

Abstract

This thesis introduces techniques to visualize information of the spatial feed data for the online application of the Swiss Feed Database. Since the computation of the Kernel methods that are used to compute the visualized images is not very time efficient, we fight the challenge to develop an algorithm of lower complexity, by introducing some optimizations: query optimization, sparse grid with bilinear interpolation and server side implementation. The experimental evaluation proves that the implemented algorithm executes fast on all popular browsers, no matter what criteria were selected by the user.

Zusammenfassung

Diese Bachelorarbeit behandelt eine Technik zur Darstellung der räumlichen Verteilung von Nahrungsmitteldaten auf der Online-Applikation der Schweizerischen Futtermitteldatenbank. Weil die zur Berechnung verwendeten Kernel Methoden nicht sonderlich zeiteffizient ausgeführt werden können, war es eine Herausforderung einen Algorithmus von geringer Komplexität zu entwickeln. Um dies zu erreichen wurden einige Optimierungen eingeführt: Eine Query-Optimierung, ein spärliches Gitter mit bilinearer Interpolation und eine serverseitige Implementierung. Die experimentelle Evaluation beweist, dass der implementierte Algorithmus in allen gängigen Browsern und unabhängig von den durch die User selektierten Kriterien schnell ausgeführt wird.

Table of Contents

1 Introduction

The feed samples of the Swiss Feed Database are collected from different locations and those are shown on the map in the application. Currently the locations from where the data samples come from are illustrated as simple flags. This has the restrictions that it is hard to visually see where the most samples come from. And moreover it is impossible to figure out where the quality of a nutrient is high.

The goal of this thesis is to enhance the online application of the Swiss Feed Database with the density information of the spatial feed data that varies depending on the search criteria. The Greedy Approach that implements the Kernels methods in a straightforward way and is executed on a client side is extremely slow with the running time measured in minutes. This is caused by two reasons. First, the Greedy Approach has quadratic complexity in the size of the density image. Second, the performance of the JavaScript degrades significantly as the amount of the data increases. With this work, we develop a real-time approach. Moreover, we extend our efficient technique to visualize the containment of the nutrients of the data samples. This allows the user to see on the map where the quality of a previously selected nutrient is high and where it is relatively low.

The runtime efficient density visualization relies on the following key optimizations. First, we reduce the complexity of the algorithm from quadratic to linear by using a sparse grid. Second, we reduce the size of the input on the SQL level. Third, we minimize the use of the JavaScript and data transfer by splitting the execution of the algorithm between the server and the client machine.

In order to show that the implementation is useful and runs in real-time, an evaluation is done. We experimentally prove that the algorithm runs hundreds of times faster than the Greedy Approach and that the running time does not rise when the number of feed sample grows. Furthermore, we show that the performance is stable independent on the browser at the client side, with a constant execution time of roughly 2 seconds.

This thesis is organized as following. In Section 2 the current online application of the Swiss Feed Database and its containing data is presented. The Kernel methods are explained in detail in Section 3. Section 4 describes the Greedy Visualization Approach and Section 5 reports the Runtime Efficient approach. In Section 6 the evaluation of the implementations is presented. Finally Section 7 concludes the thesis and presents further work that might be done.

2 The Swiss Feed Database

The Swiss Feed Database is a public service for farmers and researchers, which is provided by the Swiss Federal government agriculture, food and environmental research organization, also called Agroscope. The core data that is stored in the database are measurements of nutrients contained in feed samples. The data is derived from chemical analyses of field samples, which were collected from all over Switzerland. Currently nearly four million measurements for over 900 feed types are stored in the Swiss Feed Database.

Interface:

The Swiss Feed Database is accessible to its users over an online interface that is available in three different languages (German, English and French). It allows the users to search either for detailed data or for aggregated data. In both cases the users must select some feed types and nutrients that should be displayed. For aggregated data a table is presented after the users selection has been executed. This table shows the aggregated values for the given selection. In the case that the user selected detailed data, the user might specify its search with a further selection of a given time or a geographical restriction. When the search button is clicked, a query is executed that considers all selections that were made. The data that is retrieved from the query is then shown in different ways. For more details we consider an example.

Example:

Figure 2.1 shows how detailed data is represented in the online interface. For the example the selected feed is hay and the selected nutrients are calcium, copper and magnesium.

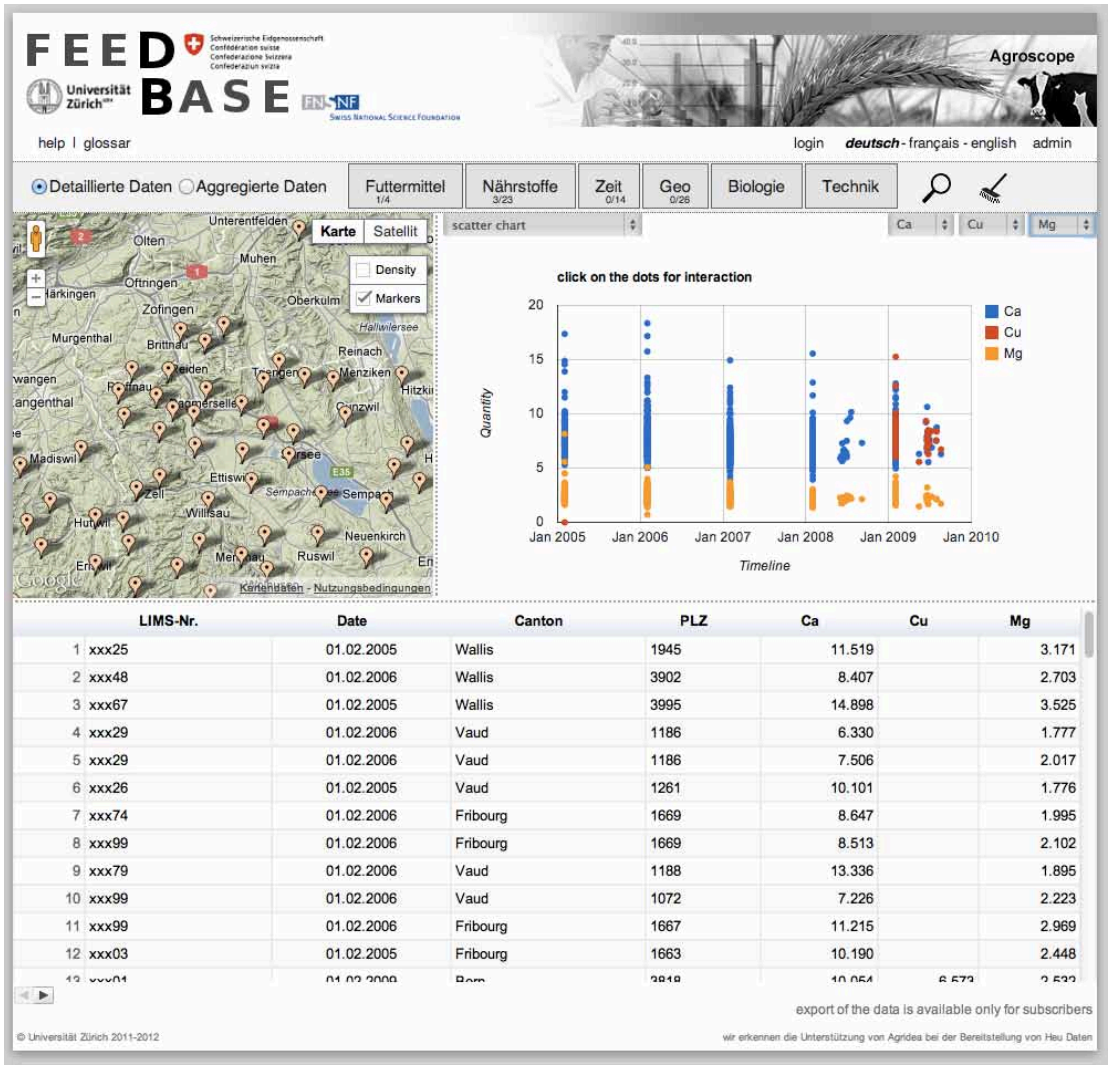


Figure 2.1: Current online representation of detailed data for hay and the nutrients calcium, cooper and magnesium.

Under “Futtermittel” the feed type hay was selected and under “Nährstoffe” the three nutrients calcium, cooper and magnesium were selected. No more restrictions were selected under “Zeit” and “Geo”, therefore the measurements from all locations and from anytime are displayed as result.

The resulting data is represented in three ways. See Figure 2.1 as an example. In the top left corner a map is displayed that as flags shows the locations, from where the feed samples were collected. Just at the right side of the map some charts can be displayed. They show the quantity of the measured nutrients in relation to the time when the samples were collected. It is possible to select at most three nutrients that can be presented in the chart. As third the data is also represented in a table. Every sample of the selected feed, for which at least for one of the selected nutrients the quantity was measured, is shown in a separate row. Besides the LIMS-number, the collection date and location, the measured quantity of the nutrients is displayed.

2.1 The Data

In this section the data of the Swiss Feed Database is described. The following mentioned data structures are stored in the database in order to run the application.

- *Feed*: a feed is mostly an agricultural foodstuff that is used to feed animals but it can also be some animal's side product such as urine or excrements. Each feed has a name, which is stored in three different languages (German, English and French). An artificial identifier is used to be able to uniquely identify a feed. Example:

id	name_de	name_en	name_fr
1	Hafer	Oats	Avoine
2	Maisstaerke	Maize starch	Amidon de mais

- *Feed Group*: a feed group groups different feeds of the same category. It contains one or more feeds and each feed can only be part of one feed group but not every feed belongs to a specific feed group. The name of the feed group can be stored in three different languages (German, English and French) and an artificial identifier is used to be able to uniquely identify a feed group. A feed group may have another feed group as its parent. Example:

id	parent_id	name_de	name_en	name_fr
1	-	Raufutter	Roughage	Fourrages
2	1	Stroh	Straw	Pailles

- *Species*: a species of animals has a name, which can be stored in three different languages (German, English and French) and an artificial identifier is used to be able to uniquely identify a specie. Example:

id	name_de	name_en	name_fr
1	Geflügel	Poultry	Volailles
2	Pferde	Horses	Chevaux

- *Nutrient*: a nutrient can either be a part of a feed or the rate of digestibility of the nutrient for a given species of animal. The name of the nutrient can be stored in three different languages (German, English and French) and a column for the abbreviation of the name is included. An artificial identifier is used to be able to uniquely identify a nutrient. Example:

id	name_de	name_en	name_fr	abbreviation
1	Kalzium	Calcium	Calcium	Ca
2	Magnesium	Magnesium	Magnésium	Mg

- *Nutrient Group*: a nutrient group groups the nutrients according to their biological properties. For example we can find nutrient groups such as carbohydrates, minerals or vitamins. A nutrient group contains one or more nutrients, its name is stored in three different languages (German, English and French) and an artificial identifier is used to be able to

uniquely identify a nutrient group. A nutrient group may be bound to an animal specie. In this case, the rate of digestibility of the different nutrients for the given species is the purpose. Example:

id	name_de	name_en	name_fr	species_id
1	Kohlenhydrate	Carbohydrates	Glucides	
2	Energiewert	Energy value	Valeurs énergétique	1

- *Sample*: a sample of a feed is taken to measure the containment of different nutrients. It stores from what laboratory the sample was prepared and an artificial identifier is used to be able to uniquely identify a sample. In addition almost every sample also has a LIMS-number, which uniquely identifies it. Example:

id	LIMS_nr	preparation
1	202850-9	LU-BR1
2	318472-7	LYO-BR1

- *Origin*: an origin is a location from where the sample might have been taken. All geographical information such us canton, postal code, city name, latitude and longitude of the locations are stored. An artificial identifier is used to be able to uniquely identify an origin. Example:

id	canton	postal_code	city	latitude	longitude
1	Luzern	6022	Grosswangen	47.13677200	8.05094400
2	Bern	3123	Belp	46.89525200	7.50462900

- *Time*: a time is a definition of the moment when the samples were taken. The exact date, the year, the month and the season in three different languages (German, English and French) are stored. An artificial identifier is used to be able to uniquely identify a time definition. Example:

id	day	year	month	season_de	season_en	season_fr
1	2001-09-21	2001	9	Herbst	Autumn	-
2	2006-09-12	2006	9	Sommer	Summer	-

- *Measurement*: a measurement stores the chemical analyzed quantity of a certain nutrient in a particular sample. In addition the origin of the sample, the time when the sample was taken and the feed type is stored as well as an artificial identifier, which ensures that a measurement can uniquely be identified. Example:

id	quantity	nutrient	sample	origin	time	feed
1	1.5854	2	1	1	2	2
2	6.6	1	2	2	1	2

The data in the Swiss Feed Database has the properties that it is rather sparse. This means that for many samples it is common that not all information is stored. But for this thesis we only consider the measurements for which complete geographical information respectively quantity is available.

Currently the number of stored geographical locations, which are possible origins of the collected feed samples, lays around 1510. As we will see, this number has an important impact on this thesis.

2.2 The Database

In this section the database that stores the data, which was described in the section above, is described. The central table of the Swiss Feed Database is the fact_table. For every measurement that was taken within a sample an entry is created. For further information the fact_table contains many foreign keys to other tables that contain detailed information about the sample, the time when the sample was collected, the location where it was collected, its containing nutrient and from what feed it is. Beside the fact_table the origin_table is important for this project, because it contains information such as city name, postal code, canton and the geographical coordinates as longitude and latitude for each possible location, where samples are collected.

Figure 2.2 shows the model of the slightly reduced Swiss Feed Database:

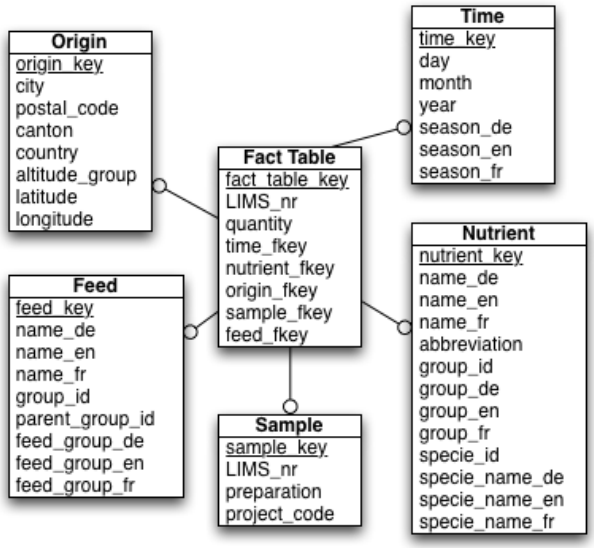


Figure 2.2: Model of the Swiss Feed Database.

In this report a simplified example of the database is used to illustrate the concepts of the implementation. The following four Tables show the example:

Table 2.1: Example fact_table table.

f_t_key	LIMS_nr	quantity	time_fkey	nutrient_fkey	origin_fkey	sample_fkey	feed_fkey
1	01-1	6.2	1	2	3	1	1
2	01-2	7.4	2	2	2	2	1
3	01-2	3.9	2	3	2	2	1
4	01-3	3.7	3	3	4	3	1
5	01-4	6.5	4	1	1	4	1
6	01-4	4.0	4	3	1	4	1
7	01-4	6.3	4	2	1	4	1
8	01-5	8.1	5	2	3	5	1

Table 2.2: Example origin table.

origin_key	city	postal_code	canton	country	a_g	latitude	longitude
1	Hergiswil NW	6052	Nidwalden	Switzerland	> 1000	46.99	8.28
2	Unterwasser	9657	St. Gallen	Switzerland	> 1000	47.56	8.89
3	Ennetbühl	9651	St. Gallen	Switzerland	> 1000	47.24	9.27

4	Andermatt	6490	Uri	Switzerland	> 1000	46.63	8.62
5	Spiringen	6464	Uri	Switzerland	> 1000	46.87	8.74

Table 2.3: Example feed table.

feed_key	name_de	name_en	name_fr
1	Heu / Emd gemischt	Hay all cuts	Foin / Regain mélange

Table 2.4: Example nutrient table.

nutrient_key	name_de	name_en	name_fr	abbreviation
1	Kalzium	Calcium	Calcium	Ca
2	Kupfer	Cooper	Cuivre	Cu
3	Magnesium	Magnesium	Magnésium	Mg

The example includes eight measurements of the nutrients calcium, cooper respectively magnesium. Those measurements were collected from four different locations and within five different samples. All samples were taken from the feed type hay. The origin table stores geographical information about five possible locations, from where samples could be collected. But only from four of those measurements are available.

2.4 The Application

The database for the application is set up on a PostgreSQL system, which is a free and open source software. It is mostly conform with the SQL programming language. On the server side the application runs with PHP scripts, which allows to dynamically load the content of the website. PHP also has built in methods to get a connection to the PostgreSQL database and to request and receive data.

On the client side the application runs by displaying HTML code in all common browsers. Executing JavaScript scripts dynamically changes this code and offers the users the possibility to interact with the system. Some JavaScript API's are used in order to be able to give the users a greater experience when using the application. The JQuery Library is used to simplify the client-side scripting and to support event handling. Moreover the Google Maps API is needed to present maps to the users. This API allows integrating maps into the application in a simple way. And finally for the charts and tables that are displayed on the application, it makes use of the Google Visualization API.

Ajax techniques are used to asynchronously load content into the application. These allow the system to send requests to the server without interfering with the currently displayed site and dynamically load the received data into it. XMlHttpRequest objects respectively ActiveXObjects are used to do this, depending on the browser that is used by the client. Both of them are API's that are available for JavaScript. The PHP scripts that are executed using Ajax also need to return some data. The Document Object Model, so called DOM, usually does this. According to [3] DOM is an interface that is language-independent and allows representing data in HTML respectively XML format. Because DOM objects can easily be read and produced in both, PHP and JavaScript, it is a convenient option to use it for data transfers.

Figure 2.3 shows a schema of how the application runs.

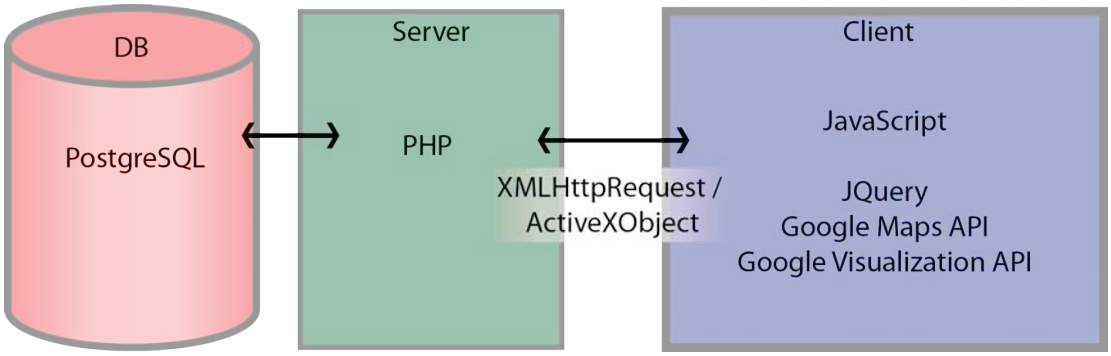
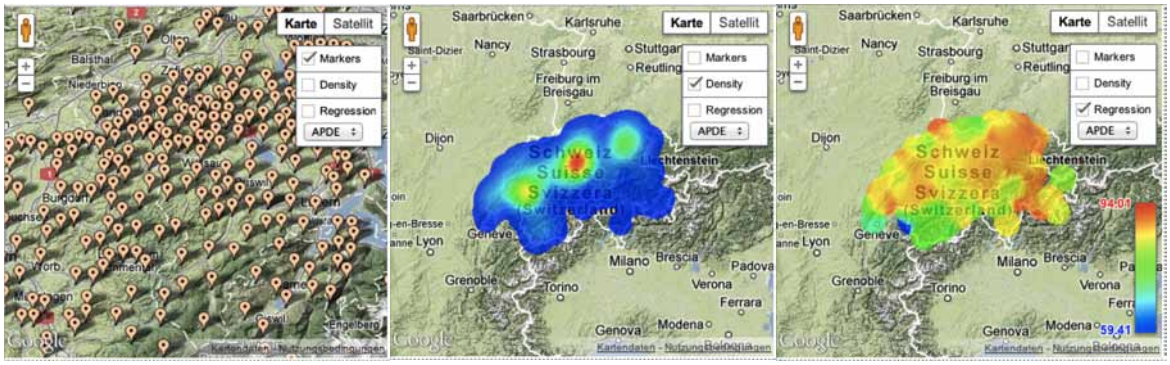


Figure 2.3: Application schema of the Swiss Feed Database.

3 Density Visualization

In the online application of the Swiss Feed Database the feed samples are illustrated on the map using flags. Figure 3.1 (a) illustrates such a representation with flags for all hay samples that contain measurements of the nutrient "Absorbierbares Protein Darm". The essential drawback of this method is, that it does not allow the users to compare different locations based on the number of feed samples. For example, if a user wants to see from which location the most feed samples come from, it cannot be easily seen since many flags overlap and that visually results in fewer flags than the number of samples.



(a) Visualization with flags. (b) Density visualization. (c) Visualization of the "Absorbierbares Protein Darm" containment.

Figure 3.1: Visualization of feed samples.

We attack this problem with the Kernel Density Estimation. This method takes into account all feed samples of the query result and allows us to compute the expected density even in the areas that are between the flags. That results in a smooth density image, which enables the comparison of the number of samples between the regions in an easy and more intuitive way. As an example consider Figure 3.1 (b). The colored density image is placed on the map. The red color corresponds to the regions with the high number of samples and the blue color denotes regions with a low number of samples. From the given density image we can easily detect three hot spots: first, the most of the samples come from the region around the canton Luzern, second the number of samples in the cantons Fribourg and St. Gallen is high too.

Another drawback of the visualization with flags (and of the density visualization) is, that it is not possible for the users to see where the containment of the selected nutrients for the given feed type is high and where it is low. To solve this problem we use the Kernel Regression technique. Similarly to the Kernel Density Estimation, the Kernel Regression results in an image where hot colors indicate high concentration of the selected nutrient. As an example consider Figure 3.1 (c). The colored image on top of the map graphically illustrates the containment of the nutrient "Absorbierbares Protein Darm" in the collected hay samples. It can be seen, that the concentration in the hay of the mountains is slightly lower than in the hay samples of the Swiss midland.

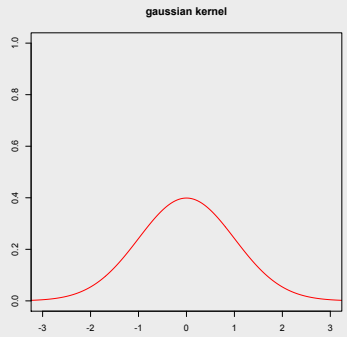
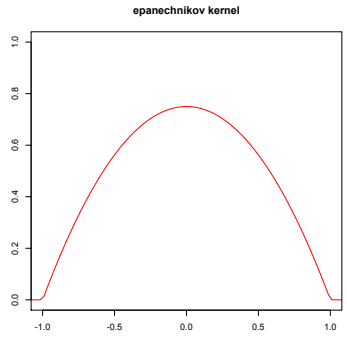
3.1 The Kernel Function

The Kernel Density Estimation and Kernel Regression rely on the Kernel Function which, in order to determine the density at an arbitrary point in space, must be evaluated for each data point in the worst case. The Kernel Function K is a symmetric function which integral is equal to 1, i.e.:

$$\int_{-\infty}^{\infty} K(x)dx = 1.$$

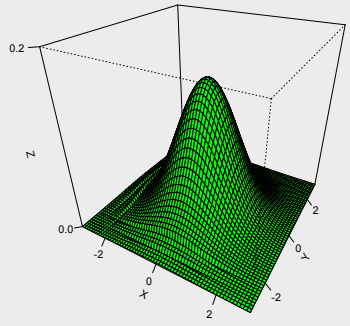
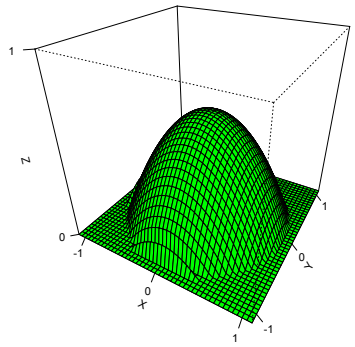
Table 3.1 shows the one-dimensional Gaussian and Epanechnikov Kernel Functions, which are according to [7] known to result in a low estimation error of the density function of the data. The shape of both Kernel Functions is a bell that is placed at point $x = 0$, i.e., the Kernel function takes the highest value at $x = 0$ and smoothly decreases down to 0 in both directions. The difference between these Kernel Functions is their computational efficiency: while the Gaussian Function requires computationally expensive exponentiation and is defined in the whole domain, the Epanechnikov Function is quadratic and, more importantly, is non-zero only in a fixed interval ($|x| \leq 1$). Therefore, in our approach we choose Epanechnikov Function and optimize computation of the density function at point x by pruning those data points for which the Epanechnikov Function is 0.

Table 3.1: Gaussian and Epanechnikov Kernel Functions.

Name	Equation	Graph
Gaussian	$K(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2}$	
Epanechnikov	$K(x) = \begin{cases} \frac{3}{4}(1-x^2), if \ \|x\| \leq 1 \\ 0, \quad otherwise \end{cases}$	

Since our main target is spatial data, we use the two-dimensional Epanechnikov Kernel Function, which fulfills all the above properties and is graphically illustrated in Table 3.2.

Table 3.2: Bivariate Gaussian and Epanechnikov Kernel Functions.

Name	**Equation**	**Graph**
Gaussian	$K(x,y) = \frac{1}{2\pi}\, e^{-\frac{x^2+y^2}{2}}$	
Epanechnikov	$K(x,y) = \begin{cases} \frac{2}{\pi}(1-x^2-y^2), & if\ x^2+y^2<1 \\ 0, & otherwise. \end{cases}$	

3.2 Kernel Density Estimation

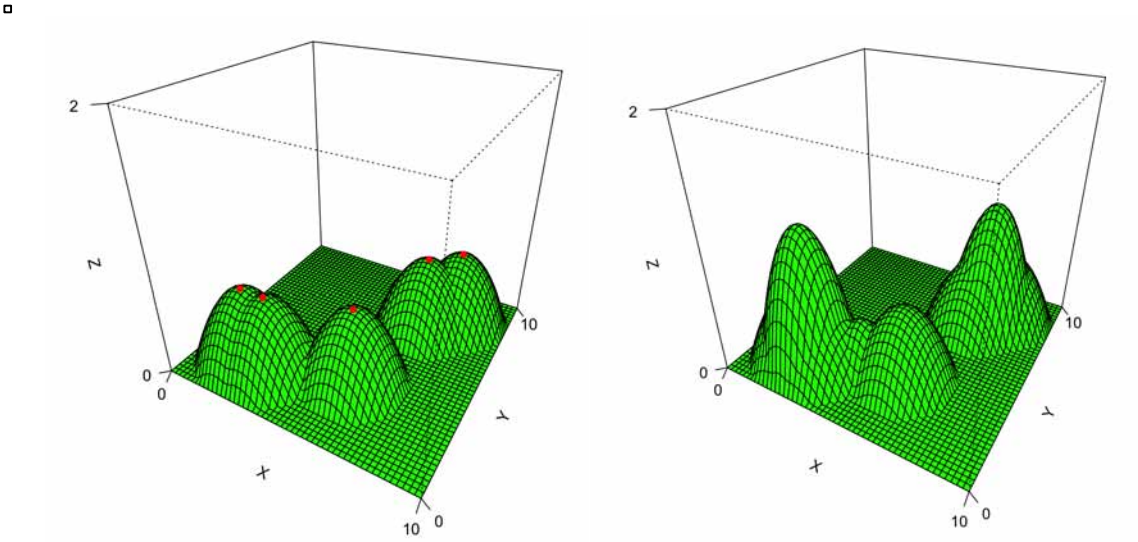
Given a finite set of data points, the aim of the Kernel Density Estimation is, to estimate the density of the whole population at every point of the domain. Differently from other estimation techniques as histograms, the Kernel Density Estimation results in a smooth density and minimizes the estimation error. The estimation error decreases as the number of data points increase.

Definition: let $L = \{(X_1, Y_1), \dots, (X_i, Y_i), \dots, (X_n, Y_n)\}$ be a set of locations with X and Y as X- and Y-Coordinates. The two-dimensional Kernel Density Estimator $\hat{f}_h(x, y)$ at the location (x, y) is:

$$\hat{f}_h(x, y) = \frac{1}{nh^2} \sum_{i=1}^{n} K\left(\frac{x - X_i}{h}, \frac{y - Y_i}{h}\right) \tag{3.1}$$

with Kernel function K and bandwidth h.

Figure 3.2 illustrates the two-dimensional Kernel Density Estimate of a data set of 5 locations ($L = \{(2,2), (3,2), (6,3), (7,7), (8,8)\}$) with a bandwidth $h = 2$.



(a) individual Kernels. (b) Kernel Density Estimation.

Figure 3.2: Two-dimensional Kernel Density Estimation.

Figure 3.2 (a) shows the individual Kernel functions placed on top of each data point. The density function is achieved by summing up these Kernel functions. The result is illustrated in Figure 3.2 (b): locations, which accumulate large number of Kernel functions results in the peaks in the density.

3.4 Kernel Regression

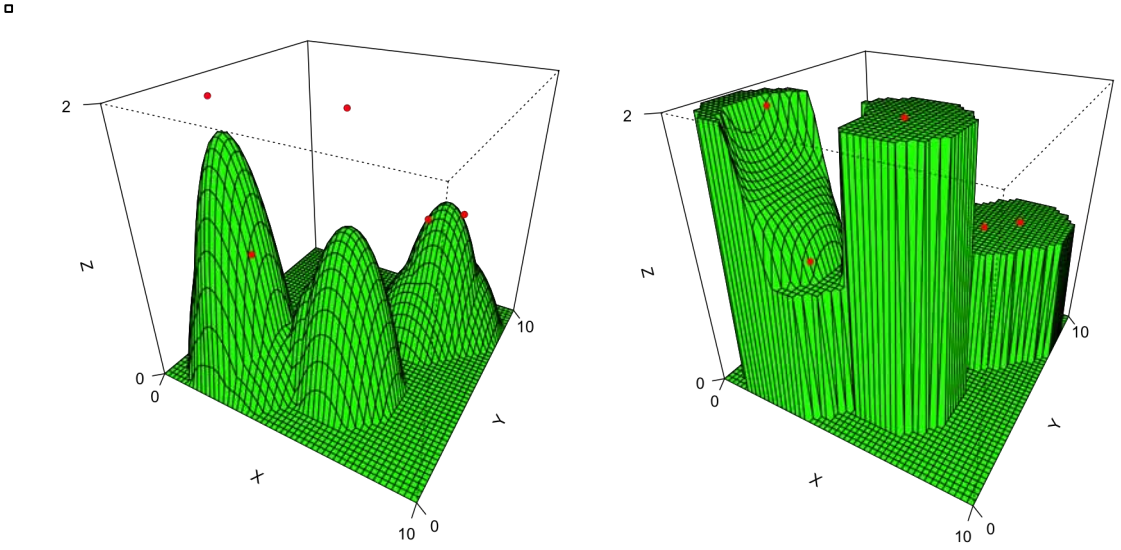
The Kernel Regression deals with data that consists of one or more independent variables and a related dependent variable. In the case of the feed data, the independent variables are locations, i.e., x and z coordinates, and the dependent variable is the measurement of nutrients at the locations. The goal of the Kernel Regression is, to estimate the value of the dependent variable for the whole population and at every possible point, i.e., the value of the nutrient at each location.

Definition: let $M = \{(X_1, Y_1, Q_1), \dots, (X_i, Y_i, Q_i), \dots, (X_n, Y_n, Q_n)\}$ be a set of measurements where X and Y are the coordinates from the location where the sample was collected and Q the measured quantity of some nutrient. Then, the two-dimensional Kernel Regression $\hat{g}_h(x, y)$ at the location (x, y) is:

$$\hat{g}_h(x, y) = \frac{\sum_{i=1}^{n} K\left(\frac{x - X_i}{h}, \frac{y - Y_i}{h}\right) Q_i}{\sum_{i=1}^{n} K\left(\frac{x - X_i}{h}, \frac{y - Y_i}{h}\right)} \tag{3.2}$$

with Kernel Function K and bandwidth h.

Figure 3.3 illustrates the two-dimensional Kernel Regression of a data set of 5 measurements ($M = \{(2,2,2), (3,2,1), (6,3,2), (7,7,1), (8,8,1)\}$) with a bandwidth $h = 2$.



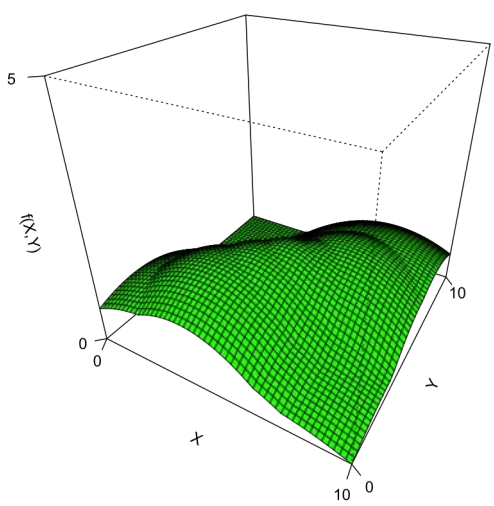
(a) weighted Kernel Density Estimation. (b) Kernel Regression.

Figure 3.3: Two-dimensional Kernel Regression.

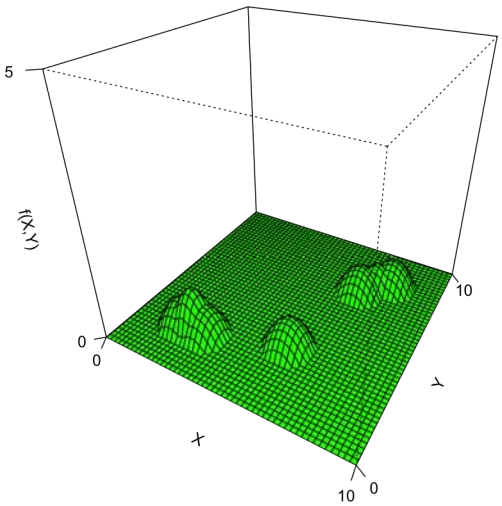
Figure 3.3 (a) shows the weighted Kernel Density Estimate, which is the numerator of Equation (3.2). And Figure 3.3 (b) represents the final Kernel Regression, which is achieved by dividing the weighted Kernel Density Estimate by the Kernel Density. It can be seen, that the Kernel Regression results on rapid jumps for locations with only few data points. Only in the regions where multiple data points are close the regression becomes smooth.

3.6 Computation of the optimal Bandwidth

The bandwidth h has a critical role in the Kernel Density Estimation. It controls the width of the base of the Kernel function. Therefore, too high values of the bandwidth results in an over smoothed density and too low values produce a density function with many oscillations as you can see in Figure 3.4.



(a) over smoothed Kernel Density Estimation with h=5.



(b) under smoothed Kernel Density Estimation with h=1.

Figure 3.4: Over and under smoothed two-dimensional Kernel Density Estimation.

The optimal bandwidth h_{opt} is computed based on the variance of the given data as following:

$$h_{opt} = \sigma * A(K) * n^{-\frac{1}{6}}$$

where σ is the variance of the data samples and $A(K)$ a constant that depends on the Kernel. For the two-dimensional Gaussian Kernel $A(K) = 0.96$ and for the tow-dimensional Epanechnikov Kernel $A(K) = 1.77$.

4 Greedy Visualization Approach

Technically, the integration of the density information into the online application of the Swiss Feed Database consists of three steps: we must estimate the density, transform it into a color image and draw this image on top of the map. In this section we present and evaluate the Greedy Approach for density visualization that involves straightforward implementation of the Kernel Density Estimator on the client side with JavaScript. The resulting performance of the Greedy Approach is strongly affected by the data size, resolution of the image and the execution speed of the JavaScript engine that varies substantially between different browsers.

4.1 Image Resolution

The first thing to consider is the resolution of the density image. Because the users usually zoom in and out of the map, it is the default requirement for the density image to have a resolution. Otherwise, the visual appearance degrades significantly when the user starts to investigate some regions in more details. With the default size of the map in the online application of the Swiss Feed Database, the image should have at least 1′000 pixels in every dimension. We call this parameter r_{image} for image resolution and, with the default settings, it results in a density image with $r_{image} * r_{image} = 1'000'000$ pixels.

4.2 Implementation

In more details, the computation of the density image consists of the following four steps:

1. ***Query execution:*** a SQL query that fetches the relevant data based on the users' selection is executed by the database. The resulting tuples are transferred to the client machine and stored in a two-dimensional array by the JavaScript script.

2. ***Calculation of the density:*** we create a temporal two-dimensional array whose size is equal to the number of pixels in the density image. Then, each element of the array represents a distinct location and for each location we scan the data and compute the density value. The additional step is to transform geographic coordinates of each origin into *xy*-coordinates in Euclidean space.

3. ***Coloring of the image:*** in this step the density image is created and the pixels are colored according to their corresponding density values in the temporal two-dimensional array.

4. ***Placing the image on the map:*** Finally the density image is placed on the map using an OverlayView of the Google Maps API.

Figure 4.1 graphically illustrates these steps for the Greedy Approach. Next, we in details explain Steps 1 and 2. The other steps (3 and 4) are explained in Section 5 together with the runtime efficient approach.

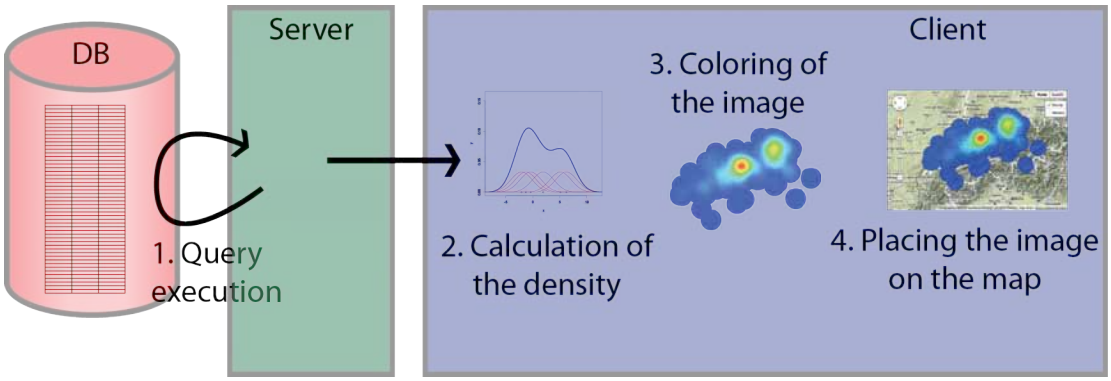


Figure 4.1: Steps of the greedy density visualization algorithm.

Query execution

When a user clicks on the button to show the density information, the first thing to execute is a PHP script on the server side that fetches all relevant data from the database. The underlying SQL query is illustrated in Example Code 4.1.

```
1:  $query = "select latitude, longitude ".
2:       "from fact_table, d_origin, d_feed, d_nutrient, d_time ".
3:       "where fact_table.id_origin_fkey = d_origin.origin_key $where";
```

Example Code 4.1

The query makes a natural join between four tables and selects latitude and longitude of all nutrient measurements that satisfy the where statement. The where statement is constructed dynamically, based on the user selections, in the online application. Table 4.1 shows the result of the SQL query for the example data (cf. Section 2) when a user puts no restrictions for nutrients, locations or the time. As we see, the output is latitude and longitude of each nutrient measurement that satisfies the search criteria. Note, that in the output we can have repetitions. For example, coordinate (47.56, 8.89) appears 2 times since there are two measurement from this location.

Table 4.1: Query result for the example data.

latitude	longitude
47.24	9.27
47.56	8.89
47.56	8.89
46.63	8.62
46.99	8.28
46.99	8.28
46.99	8.28
47.24	9.27

Calculation of the density

The implementation is done exactly as the Kernel Density Estimator is defined. For every position, in our case pixel, for which we want to have the density, the influence of every data tuple (which essentially corresponds to geographic coordinates returned by the SQL query) to this pixel has to be summed up. Example Code 4.2 shows how it is implemented.

```
 1: function epanechnikovKernel(dx,dy)
 2: {
 3:    var t = Math.pow(dx,2) + Math.pow(dy,2);
 4:    if (t >= 1)
 5:       return 0;
 6:    return 2 / Math.PI * ( 1 - t);
 7: }
 8: function computeDensity(x, y, data, hopt)
 9: {
10:    var density = 0;
11:    for (var p = 0; p < data.length; p++)
12:    {
13:       var point = data[p];
14:       density += epanechnikov_kernel((point.x-x)/hopt,(point.y-y)/hopt);
15:    }
16:    return density;
17: }
```

Example Code 4.2

On lines 1 to 7 in Example Code 4.2 the implementation of the Epanechnikov Kernel is illustrated. As an input, this function takes dx and dy, which are distances along both dimensions from the center of the Epanechnikov Kernel. On lines 8 to 17 the implementation of the Kernel Density Estimation is shown. The function takes the x and y coordinates of the considered pixel, the array with all data tuples and the optimal bandwidth as arguments. Lines 11 to 15 iterate through all data tuples. For every data tuple we compute the distance to the considered pixel and evaluate the Epanechnikov Kernel. At last, the outputs are summed up and the function returns the density of the current pixel. In order to have the density values for the whole image calculated, the computeDensity function needs to be executed for every pixel of the image.

4.3 Complexity

As we already specified, we are calculating an image that has a resolution of $r_{image} = 1'000$ pixels in each dimension and that results in $r_{image} * r_{image} = 1'000'000$ pixels. For every pixel the Kernel Density Estimator needs to be calculated. To do so every data point must be considered. If n is the number of data points, then the final complexity of the computation is: $O(r_{image}^2 * n)$. Practically, because of the high resolution, slow JavaScript and large n of more than $1'000'000$ that results in an unacceptable runtime that is measured in minutes.

5 Runtime Efficient Visualization Approach

In this section we present the critical optimizations that are necessary in order to improve the performance of the density visualization. Three steps, which improve the efficiency of the Greedy Approach, are implemented:

1. ***Optimize the SQL query:*** As the first step the SQL query is optimized. This optimization aims to reduce the number of measurements n. The idea is that measurements from feed samples, which were collected at the same location, are grouped together. Since the number of distinct locations in the Swiss Feed Database is much smaller than the number of measurements, with this optimization we are able to reduce n from about 10^6 to 10^3.

2. ***Sparse grid:*** As the second step the calculation of the density is done for a smaller number of points/pixels. Instead of computing the density for each pixel of a high-resolution image, we introduce a sparse grid that has a small number of points. Then, we compute the density for each point of the sparse grid. The transition from the sparse grid to a high-resolution image is achieved with the help of linear interpolation.

3. ***Server side implementation:*** As the third step, we shift the calculation of the density from the client machine to the server. That helps to reduce the runtime in two ways. First, we eliminate the use of the JavaScript which performance is very dependent on the browser and computer the users use. Second, we reduce the amount of the data that must be transferred (and stored) to the client machine, i.e., instead of transferring geographic coordinates for each nutrient measurement, we send only the resulting density measurement at points of the sparse grid.

5.2 Optimize the SQL query

Because many data points usually come from the same location, it is possible to optimize the query. The idea is to take all data points from the same location together and count its occurrences. Afterwards the data points can be seen as a triple of latitude, longitude and the number of occurrences. With this information the density can be computed. Example Code 5.1 shows the query that is used to fetch the data from the database.

```
1:  $query = "select latitude, longitude, count(*) ".
2:           "from fact_table, d_origin, d_feed, d_nutrient, d_time ".
3:           "where fact_table.id_origin_fkey = d_origin.origin_key $where".
4:           "group by latitude, longitude";
```

Example Code 5.1

If we have a look at the example from the beginning of this report the tuples in Table 5.1 result from a query that includes all eight available measurements.

Table 5.1: Query result for the example data.

latitude	longitude	count
46.99	8.28	3
47.56	8.89	2
47.24	9.27	2
46.63	8.62	1

With this optimization we are able to reduce the complexity of the computation. The Swiss Feed Database contains at the moment about 1510 different possible locations. We call this parameter m. So we can say the complexity is in worst-case $O(r_{image}^2 * m)$. Note that m is by hundreds of times lower than the number of measurements in the fact table n.

For the Kernel Regression we need to take a slightly different query, because we can only calculate the regression for the measurements of one nutrient. Therefore the user previously needs to select the nutrient for which the values should be visualized. The selected nutrient is then passed as parameter to the script that executes the query. The second change comes because we also need the quantity of the measurements. Therefore the query also selects the average of the quantity of the measurements at every location. So the data points finally can be seen as a quadruple of latitude, longitude, the number of occurrences and the average quantity of all data points at the given location. With this information the Kernel Regression can be computed. Example Code 5.2 shows the query that is used to fetch the data from the database.

```
1:  $query = "select latitude, longitude, count(*), avg(quantity) ".
2:           "from fact_table, d_origin, d_feed, d_nutrient, d_time ".
3:           "where fact_table.id_origin_fkey = d_origin.origin_key $where ".
4:           "and d_nutrient.z_abbreviation_de in ('$nutrient')".
5:           "group by latitude, longitude";
```

Example Code 5.2

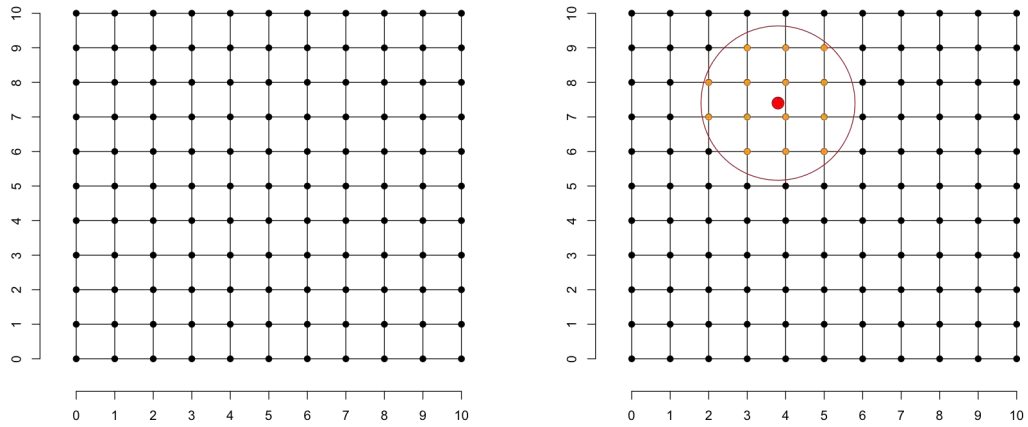
If we have a look at the example from the beginning of this report, the tuples in Table 5.2 result from a query that includes all measurements of the nutrient cooper.

Table 5.2: Query result for the example data.

latitude	longitude	count	quantity
46.99	8.28	1	6.3
47.56	8.89	1	7.4
47.24	9.27	2	7.15

5.4 Sparse grid

We now introduce a sparse grid, with which the running time of the density computation can be decreased. The idea is, that we do not calculate the density value of each pixel itself, but we calculate the density for a defined grid and finally draw an image with high resolution by using bilinear interpolation to derive the density values within the grid points.



(a) simple grid. (b) grid with one data point.

Figure 5.1: Sparse Grid.

When we use a sparse grid we have the advantage that every data point only has an influence on a few grid points. Figure 5.1 (b) shows in red a data point for which the contribution to the density of the grid points should be calculated. This data point has actually only some influence on the orange grid points that lay within the red circle. The red circle has a radius of h_{opt}.

If a grid is used that is very sparse the number of grid points that need to be considered for the density contribution of a data point decreases significantly. But to get a resulting high-resolution image that has a good enough quality, the grid cannot be infinitely sparse. To get a good balance, we choose the size of the grid based on the value of h_{opt}, so that within a circle of radius h_{opt} there are at most 20 grid points per dimension. Such an approach leads to the following linear algorithm for density computation.

Example Code 5.3 shows the algorithm that is used to calculate the density values of all grid points.

```
 1: function computeDensity($data, $height, $width, $hopt)
 2: {
 3:     $densities = array_fill(0, $height, array_fill(0, $width, 0));
 4:     for ($p = 0; $p < count($data); $p++)
 5:     {
 6:         $point = $data[$p];
 7:         for ($i = $point['y'] - $hopt; $i <= $height && $i <= $point['y'] + $hopt; $i++)
 8:         {
 9:             for ($j = $point['x'] - $hopt; $j <= $width && $j <= $point['x'] + $hopt; $j++)
10:             {
11:                 if ($i > 0 && $j> 0)
12:                 {
13:                     $x = ($point['x']-$j)/$hopt;
14:                     $y = ($point['y']-$i)/$hopt;
15:                     $t = $x*$x + $y*$y;
16:                     if ($t <= 1)
17:                     {
18:                         $densities[$i-1][$j-1] = $densities[$i-1][$j-1]
                                    + (2 / pi() * ( 1 - $t)) * $point['count'];
19:                 }
20:             }
21:         }
22:     }
23:     return $densities;
24: }
```

Example Code 5.3

The function shown in Example Code 5.3 takes the whole data set, the height and width of the grid and the optimal bandwidth as arguments. On line 3 a two-dimensional array, which contains at the beginning a 0 for every grid point, is created. Afterwards on lines 4 to 22 an iteration through all data points is done. For every data point we go to the data points that lay at most h_{opt} away in each dimension. This is first, on line 7, done for the vertical dimension and afterward on line 9 for the horizontal dimension. We also check if the grid point still lies within the allowed bounds. On lines 13 to 15 the exact distance of the current data point to the current grid point is normalized with respect to h_{opt}. If this value is at most 1 the Epanechnikov Kernel is calculated for this distance and the value is summed up to the value of the currently considered grid point. This is done on line 18. Because we might have multiple data points at the currently considered location we also multiply the contribution by the number of data points. Finally, the array with the calculated density values is returned by the function on line 23.

The algorithm to calculate the Kernel Regression needs to be slightly different. It is listed in Example Code 5.4.

```
 1: function computeRegression($data, $height, $width, $hopt)
 2: {
 3:     $regressionNumerator = array_fill(0, $height, array_fill(0, $width, 0));
 4:     $regressionDenominator = array_fill(0, $height, array_fill(0, $width, 0));
 5:     $regression = array_fill(0, $height, array_fill(0, $width, 0));
 6:     for ($p = 0; $p < count($data); $p++)
 7:     {
 8:         $point = $data[$p];
 9:         for ($i = $point['y'] - $hopt; $i <= $height && $i <= $point['y'] + $hopt; $i++)
10:         {
11:             for ($j = $point['x'] - $hopt; $j <= $width && $j <= $point['x'] + $hopt; $j++)
12:             {
13:                 if ($i > 0 && $j> 0)
14:                 {
15:                     $x = ($point['x']-$j)/$hopt;
16:                     $y = ($point['y']-$i)/$hopt;
17:                     $t = $x*$x + $y*$y;
18:                     if ($t <= 1)
19:                     {
20:                         $value = (2 / pi() * ( 1 - $t)) * $point['count'];
21:                         $regressionNumerator[$i-1][$j-1] += $value * $point['quantity'];
22:                         $regressionDenominator[$i-1][$j-1] += $value;
23:                     }
24:                 }
25:             }
26:         }
27:     }
28:     for ($i = 0; $i < count($regression); $i++)
29:     {
30:         for ($j = 0; $j < count($regression[$i]); $j++)
31:         {
32:             if ($regressionDenominator[$i][$j] != 0)
33:             {
34:                 $regression[$i][$j] = $regressionNumerator [$i][$j]
                                                    / $regressionDenominator[$i][$j];
35:             }
36:         }
37:     }
38:     return $regression;
39: }
```

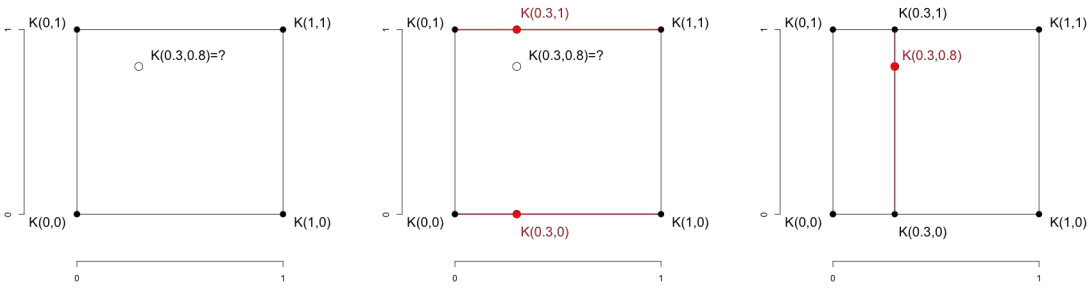
Example Code 5.4

On lines 3 and 4 the first differences to the density calculation algorithm occur. In addition to the array that is returned, two other temporary arrays are initialized. One of those will contain the values of the numerator of the Kernel Regression formula and the other one the values of the denominator. Afterwards the same iteration, through all data points, is executed. The only difference is, that on lines 20 to 22 the values of the numerator and denominator are summed up. And afterwards on lines 28 to 37 we iterate through all grid points. This is done because on line 34 the regression values are calculated by dividing the value of the numerator array through the value of the denominator value. Finally, on line 38 the array that contains the regression values for all grid points is returned.

Let us have a look at the complexity of these algorithms. Since the size of the grid is fixed based on the h_{opt}, then for every data point, which number is by now at most m, we need to consider up to a constant number of grid points. Therefore, that results on the linear time complexity $O(m)$.

Bilinear Interpolation:

We now have the density values at all grid points calculated. But, because we must create a high-resolution image we need to have the density value at every pixel between the grid points. We use bilinear interpolation to derive the missing density values.



(a) Initial state. (b) Horizontal interpolation. (c) Vertical interpolation.

Figure 5.2: Bilinear interpolation.

To simplify the definition of the bilinear interpolation we assume, that each cell has his bottom left corner at (0,0) and its top right corner at (1,1). For all four corners of the grid cell the density value $\hat{f}_h$ is known. But now we want to derive the density value for every (x, y) *with* $x \in [0,1]$ *and* $y \in [0,1]$. To do this we first interpolate in the horizontal dimension. The density values at $(x, 0)$ and $(x, 1)$ are calculated as follows:

$$\hat{f}_h(x,0) = \hat{f}_h(0,0) * x + \hat{f}_h(1,0) * (1 - x)$$
$$\hat{f}_h(x,1) = \hat{f}_h(0,1) * x + \hat{f}_h(1,1) * (1 - x)$$

With those two values we are able to interpolate in the vertical dimension. We finally define the density value at position (x, y) as:

$$\hat{f}_h(x,y) = \hat{f}_h(x,0) * y + \hat{f}_h(x,1) * (1 - y) = \hat{f}_h(x,0)$$
$$= \hat{f}_h(0,0) * xy + \hat{f}_h(1,0) * (1 - x) * y + \hat{f}_h(x,1) + \hat{f}_h(0,0) * x$$
$$* (1 - y) + \hat{f}_h(1,1) * (1 - x) * (1 - y)$$

Let us consider an example. We want to calculate the density value at position $(x, y) = (0.3,0.8)$ and we take the values of the grid points as $\hat{f}_h(0,0) = 0.2; \hat{f}_h(1,0) = 0.7; \hat{f}_h(0,1) = 0.6; \hat{f}_h(1,1) = 0.3$. From $\hat{f}_h(0,0)$and $\hat{f}_h(1,0)$ we get an interpolated density value at position $\hat{f}_h(0.3,0)$:

$$\hat{f}_h(0.3,0) = 0.2 * 0.3 + 0.7 * (1 - 0.3) = 0.55.$$

From $\hat{f}_h(0,1)$and $\hat{f}_h(1,1)$ we get an interpolated density value at position $\hat{f}_h(0.3,1)$:

$$\hat{f}_h(0.3,1) = 0.6 * 0.3 + 0.3 * (1 - 0.3) = 0.39.$$

Finally we linearly interpolate in the vertical dimension between $\hat{f}_h(0.3,0)$and $\hat{f}_h(0.3,1)$ and get the density value at position $\hat{f}_h(0.3,0.8)$:

$$\hat{f}_h(0.3,0.8) = 0.55 * 0.8 + 0.39 * (1 - 0.8) = 0.518.$$

5.6 Server side implementation

The aim is to execute the most time-consuming parts of the implementation on the server rather than on the client side. That helps us to avoid using slow JavaScript to calculate the density and substantially reduces the data transfer. Since the data required computing the density does not need to be transferred, but only the calculated density values of the grid points.

The critical steps of the implementation are the following:

1. ***Query execution:*** the query that fetches the relevant data from the database needs to be executed.

2. ***Data transformation:*** the coordinates of the data samples are retrieved as longitude and latitude, but to compute the density image the position of the data samples on the image need to be calculated.

3. ***Calculation of the density:*** in this step the density value for every grid point is calculated. This step includes the calculation of the optimal bandwidth h, which is needed to calculate the density.

4. ***Density transfer:*** the calculated density values for the grid points are transferred from the server side to the client side.

5. ***Coloring of the image:*** in this step the image is created and the pixels are colored according to their corresponding density values. In order to get a high-resolution image also the bilinear interpolation is done in this step.

6. ***Placing the image on the map:*** Finally the image is placed on the map using an OverlayView of the Google Maps API.

Query execution

When the users first click on the button, which starts the computation of the density visualization, a PHP script that fetches the relevant data from the database is executed. The code that is used to do this is listed in Example Code 5.5.

```
 1: $where = getWhereStatement();
 2:
 3: query = "select latitude, longitude, count(*) ".
 4:           "from fact_table, d_origin, d_feed, d_nutrient, d_time ".
 5:           "where fact_table.id_origin_fkey = d_origin.origin_key $where".
 6:           "group by latitude, longitude";
 7:
 8: // Connection to PostgreSQL-DB
 9: $conn = pg_connect("host=$host port=$port dbname=$database user=$username password =
                  $password") or die  ('Error connecting to Postgres');
10: if (!$conn) die('Not connected : ' . pg_error());
11:
12: $result = pg_query($conn, $query);
13: $numrows = pg_numrows($result);
14: $tuples = array();
15:
16: for($ri = 0; $ri < $numrows; $ri++) {
17:      $row = pg_fetch_array($result, $ri);
18:      array_push($tuples, $row);
19: }
20:
21: pg_close($conn);
```

Example Code 5.5

The query defined on lines 3 to 6 is the optimized query, which was discussed in section 5.2. in detail. It selects latitude and longitude and some additional information of all data samples that satisfy the where statement. This where statement considers all selections the user took, in order to only select the relevant tuples. The connection to the PostgreSQL database is established on line 9 and again closed on line 21. Errors that may occur because of connecting problems are handled on line 10. On line 12 the defined query is executed by calling the pg_query() function. From line 16 to line 19 the resulting tuples of the SQL query are handled, by iterating through all tuples and storing their values in the previously defined tuples array. Finally we have a tuples array, which contains the latitude, longitude, count and possibly quantity of all tuples.

Data transformation

The data points that are received from the database in form of longitude and latitude need to be transformed into the position they have on the map. This is a challenging task since the earth is a three-dimensional sphere, but we need to represent it on a two-dimensional plane. This leads to the fact that, the same difference of latitudes does not lead to the same distance at every point of the map.

As an example consider Figure 5.3. The red rectangle covers an area on the map, which is 10 degree larger along each side than the yellow rectangle. As we see in the following figure the same difference of 10 degrees leads in the north to a much larger distance than around the equator. Because of this problem we cannot use latitude and longitude for computation of distances on the map in a direct way.

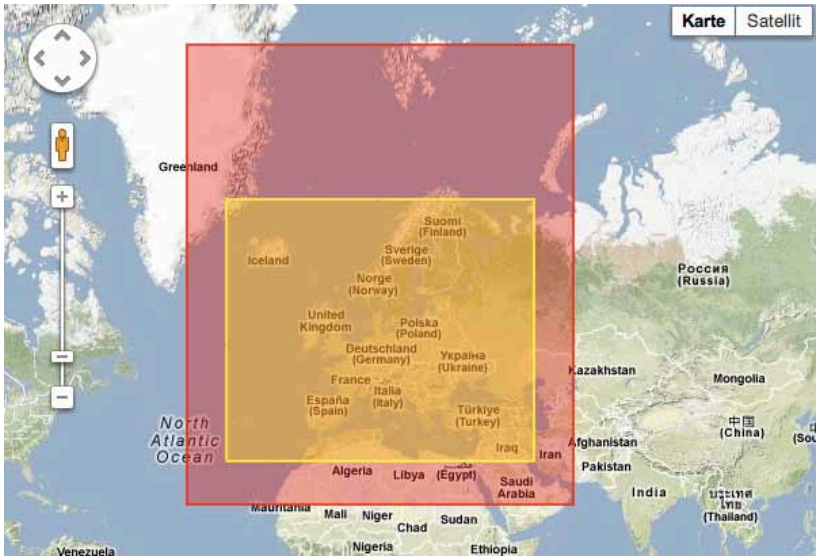


Figure 5.3: Two-dimensional sphere representation.

We solve this problem by using the Mercator projection to transform the data points to their position on the map. The following formulas are used to calculate the x and y coordinates of a point on the Mercator map from its latitude φ and longitude λ:

$$x = \lambda - \lambda_0$$

$$y = -\frac{1}{2} ln\left(\frac{1 + \sin\varphi}{1 - \sin\varphi}\right)$$

with $\lambda_0$as longitude of the center of the map [1].

As an example we consider the location with latitude $\varphi = \frac{46.99}{360} * 2\pi$ and longitude $\lambda = \frac{8.28}{360} * 2\pi$ and $\lambda_0 = 0$. With this we can calculate the x and y coordinates of this location on the map, which are:

$$x = \frac{8.28}{360} * 2\pi - 0 = 0.14451 \ldots$$

$$y = -\frac{1}{2} ln\left(\frac{1 + \sin(\frac{46.99}{360} * 2\pi)}{1 - \sin(\frac{46.99}{360} * 2\pi)}\right) = 0.93137 \ldots$$

The Example Code 5.6 shows how it is implemented. We slightly adjusted the calculation, so that the distance around the globe has a value of 1 and that the center of the map (latitude $\varphi = 0$ and longitude $\lambda = 0$) is at 0.5 in both dimensions.

```
 1: function bound($value, $opt_min, $opt_max) {
 2:    if ($opt_min != null) $value = max($value, $opt_min);
 3:    if ($opt_max != null) $value = min($value, $opt_max);
 4:    return $value;
 5: }
 6:
 7: function fromLatLngToPoint($lat, $lng)
 8: {
 9:    $center["x"] = 1/2;
10:    $center["y"] = 1/2;
11:    $distancePerDegree = 1/360;
12:    $distancePerRadian = 1/(2*pi());
13:    $x = $center["x"] + $lng * $distancePerDegree;
14:    $siny = bound(sin(deg2rad($lat)), -0.9999, 0.9999);
15:    $y = $center["y"] - 0.5 * log((1+$siny)/(1-$siny)) * $distancePerRadian;
16:    return array("x" => $x, "y" => $y);
17: }
18: function fromPointToLatLng ($x, $y )
19: {
20:    $center["x"] = 1/2;
21:    $center["y"] = 1/2;
22:    $distancePerLonDegree = 1/360;
23:    $distancePerLonRadian = 1/(2*pi());
24:    $lng = ($x - $origin['x']) / $distancePerLonDegree;
25:    $latRadians = ($y - $origin['y']) / -$distancePerLonRadian;
26:    $lat = rad2deg(2 * atan(exp($latRadians)) - pi() / 2);
27:    return array('lat' => $lat, 'lng' => $lng);
28: }
29:
30: $data = array();
31: for($t = 0; $t < count($tuples); $t++)
32: {
33:    $point = fromLatLngToPoint($tuples[$t]['latitude'], $tuples[$t]['longitude']);
34:    $point['count'] = $tuples[$t]['count'];
35:    $point['quantity'] = $tuples[$t]['quantity'];
36:    array_push($data, $point);
37: }
```

Example Code 5.6

The function bound, which is implemented on lines 1 to 5 takes a value and the minimum (respectively maximum) as arguments and returns the value or the maximum (respectively minimum), if the value lays outside of these bounds. The fromLatLngToPoint function transforms the given latitude and longitude into x and y coordinates as described in the formulas above. It returns an array containing those coordinates. The function fromPointToLatLng that is implemented on lines 18 to 28 does exactly the opposite. It transforms x and y coordinates back to latitude and longitude. This function will be used later in the implementation. On lines 31 to 37 the transformation of the data points is executed. The *for loop* iterates through all tuples in the tuples array and the transformed coordinates are appended in the newly created data array. The output of this code for the example data is illustrated in Table 5.3.

Table 5.3: Transformation result of the example data.

latitude	longitude
46.99	8.28
47.56	8.89
47.24	9.27
46.63	8.62

→

x	y
0.523	0.35177
0.52469	0.34943
0.52575	0.35075
0.52394	0.35323

Density calculation

The density at the grid points is calculated as described in the previous section. But before this we must derive the value for the optimal bandwidth. The formula that we are using is as following:

$$h_{opt} = \sigma * A(K) * n^{-\frac{1}{6}}$$

with $A(K) = 1.77$ because we are using the Epanechnikov Kernel.

To calculate the variance we take the following formula:

$$\sigma = \sqrt{\frac{\sum_{i=0}^{n} x_i^2 - \frac{\left(\sum_{i=0}^{n} x_i\right)^2}{n}}{n-1} + \frac{\sum_{i=0}^{n} y_i^2 - \frac{\left(\sum_{i=0}^{n} y_i\right)^2}{n}}{n-1}}{2}.$$

It allows us to calculate the variance with one scan of the data points. This is because all sums in the formula can be computed in parallel by iterating through all data points.

Example Code 5.7 shows how the computation of the optimal bandwidth is implemented.

```
 1: function optimalBandwidth($data)
 2: {
 3:    $n = 0;
 4:    for ($p = 0; $p < count($data); $p++)
 5:    {
 6:       $xweight += ($data[$p]['x'] * $data[$p]['count']);
 7:       $yweight += ($data[$p]['y'] * $data[$p]['count']);
 8:       $xqweight += (pow($data[$p]['x'], 2) * $data[$p]['count']);
 9:       $yqweight += (pow($data[$p]['y'], 2) * $data[$p]['count']);
10:       $n += $data[$p]['count'];
11:    }
12:    $sx = ($xqweight - ($xweight * $xweight / $n)) / ($n-1);
13:    $sy = ($yqweight - ($yweight * $yweight / $n)) / ($n-1);
14:    $sigma = sqrt( ($sx + $sy) /2);
15:
16:    return $sigma* 1.77 * pow($n, -1/6);
17: }
```

Example Code 5.7

The function optimalBandwidth takes as input the array that contains all transformed data points and returns the optimal bandwidth for those data points. On lines 4 to 11 we iterate through all data points and calculate the number of data points n, the sum of x and y coordinates as well as the sum of the square of x and y coordinates. From lines 12 to 14 the rest of the variance computation is executed and finally on line 16 the optimal bandwidth is calculated and returned.

Now, that we have the optimal bandwidth we can calculate the exact size of the grid and compute the density values at the grid points. Example Code 5.8 shows how this step is implemented.

```
 1: $hopt = optimalBandwidth($data);
 2: $height = 0; $width = 0;
 3:
 4: $minx = 1; $miny = 1; $maxx = 0; $maxy = 0;
 5: for ($p = 0; $p < count($data); $p++)
 6: {
 7:     $minx = min($data[$p]["x"], $minx);
 8:     $miny = min($data[$p]["y"], $miny);
 9:     $maxx = max($data[$p]["x"], $maxx);
10:     $maxy = max($data[$p]["y"], $maxy);
11: }
12:
13: $gridResolution = floor(($maxx-$minx+2*$hopt) / $hopt * 20);
14: if ($maxx - $minx > $maxy - $miny)
15: {
16:     $width = $gridResolution;
17:     $height = floor($gridResolution/($maxx-$minx+2*$hopt)*($maxy-$miny+2*$hopt));
18: } else {
19:     $width = floor($gridResolution/($maxy-$miny+2*$hopt)*($maxx-$minx+2*$hopt));
20:     $height = $gridResolution;
21: }
22: for ($p = 0; $p < count($data); $p++)
23: {
24:     $data[$p]['x'] = ($data[$p]['x']-$minx+$hopt) / ($maxx-$minx+2*$hopt) * $width;
25:     $data[$p]['y'] = ($data[$p]['y']-$miny+$hopt) / ($maxy-$miny+2*$hopt) * $height;
26: }
27:
28: $densities = computeDensity($data, $height, $width, $hopt);
29: $densities = computeRegression($data, $height, $width, $hopt);
```

Example Code 5.8

On the first line of the Example Code 5.8 the execution of the calculation of the optimal bandwidth is done. On lines 4 to 11 the extreme values in both dimensions are calculated with one scan of all data points. Then on line 13 the grid resolution is set depending on the size of the optimal bandwidth. Finally the grid resolution is such that the optimal bandwidth covers at most 10 grid points, this means that a circle of radius h_{opt} at most covers 20 grid points per dimension Afterwards on lines 14 to 21 the height and width of the grid are calculated. The number of points in the dimension in which the distance of the extreme values is greater is set to the calculated grid resolution and the number of points in the other dimension is calculated in order to keep the distance ratio. As next step, the data points are converted so that the x and y coordinates now correspond to the position the location has on the grid. This is done on lines 22 to 26 by iterating through all data points. Finally, on lines 28 or 29 the density values are calculated as seen in the previous section. Either line 28 is active, when the density should be calculated or line 29 is executed when the aim is to compute the regression.

Density transfer

When the density values of all points of the grid are calculated, the values must be sent to the client side. A DOM object is created to pass the density values to the JavaScript. Also the position and the size of the calculated density image need to be transferred. Example Code 5.9 shows the PHP code that is needed to transfer the information.

```
 1: $dom = new DOMDocument("1.0", "UTF-8");
 2: $node = $dom->createElement("data");
 3: $parnode = $dom->appendChild($node);
 4:
 5: $sw = fromPointToLatLng($minx-$hopt, $maxy+$hopt);
 6: $node = $dom->createElement('sw');
 7: $newnode = $parnode->appendChild($node);
 8: $newnode->setAttribute("lng", $sw["lng"]);
 9: $newnode->setAttribute("lat", $sw["lat"]);
10:
11: $ne = fromPointToLatLng($maxx+$hopt, $miny-$hopt);
12: $node = $dom->createElement('ne');
13: $newnode = $parnode->appendChild($node);
14: $newnode->setAttribute("lng", $ne["lng"]);
15: $newnode->setAttribute("lat", $ne["lat"]);
16:
17: $values = $dom->createElement('values');
18: $newnode = $parnode->appendChild($values);
19: $newnode->setAttribute("max", $max);
20: $newnode->setAttribute("min", $max);
21:
22: for ($i = 0; $i < count($densities); $i++)
23: {
24:    $row = $dom->createElement('row');
25:    $newnode = $values->appendChild($row);
26:    $newnode->setAttribute("values", implode(";", $densities[$i]));
27: }
28: echo $dom->saveXML();
```

Example Code 5.9

The DOM object is organized into nodes. In this implementation we have four different node types. We have an ‘sw’ node, which contains the information about the southwest corner of the density image. It has the latitude and longitude of this location set as attributes. Similarly the ‘ne’ node contains the latitude and longitude of the location at the northeast corner of the density image. Moreover a ‘values’ node is used. This node has the maximum and minimum density value stored as attributes and it contains a number of child nodes of the type ‘row’. The ‘row’ nodes contain the density values of one row of the grid. The values are separated with a semicolon. For every row of the grid a ‘row’ node is appended as child to the ‘values’ node.

On line 1 of the Example Code 5.9 a DOM object is initiated. This is used to create an XML document that contains the data, which need to be transferred. The data is organized as described in the previous paragraph. And finally on line 28 the created XML document is put out as echo of the PHP script. This echo is afterwards read and used by the JavaScript script. Example Code 5.10 shows an example of how the resulting XML document can look like.

```
<?xml version="1.0" encoding="UTF-8"?>
<data>
        <sw lng="5.80287078539" lat="45.7798785199"></sw>
        <ne lng="10.4993332146" lat="47.8538699134"></ne>
        <values max="5.94981317995" min="3.9824578685">
                <row
                        values="0;5.56;5.56;..."></row>
                <row
                        values="5.58736600765;5.632839196;5.56743431818;..."></row>
                <row
                        values="5.57152163631;5.58022641738;5.58854749799;..."></row>
                ...
</values>
</data>
```

Example Code 5.10

On the client side the sent data is retrieved reading the XML response of the httpObject. All information is read from the response and stored into the own variables. The elements are found in the response by searching for the corresponding tag names. As the density values were stored as semicolon separated strings they are again split and put into an array.

```
var resultXML = httpObject.responseXML.documentElement;

var swBound = new
        google.maps.LatLng(resultXML.getElementsByTagName("sw")[0].getAttribute("lat"),
        resultXML.getElementsByTagName("sw")[0].getAttribute("lng"));
var neBound = new
        google.maps.LatLng(resultXML.getElementsByTagName("ne")[0].getAttribute("lat"),
        resultXML.getElementsByTagName("ne")[0].getAttribute("lng"));
bounds = new google.maps.LatLngBounds(swBound, neBound);

var max = resultXML.getElementsByTagName("values")[0].getAttribute("max");
var min = resultXML.getElementsByTagName("values")[0].getAttribute("min");

var valuesArray = resultXML.getElementsByTagName("row");
var valuesLength = resultXML.getElementsByTagName("row").length;
var values = new Array();
for (var i = 0; i < valuesLength; i++)
{
   var valuesString = valuesArray[i].getAttribute("values");
   var newValues = valuesString.split(";");
   values.push.apply(values, newValues);
}
```

Example Code 5.11

Coloring of the image

Once the array that contains the density values for every point of the grid is retrieved at the client side, the coloring of the image can be done. Therefore a color palette is needed. The grid point with the highest density value is colored red, average density values correspond to the green color and the lowest density values are colored blue. Figure 5.4 shows the color palette, which is used for the implementation.

Figure 5.4: Color palette used to visualize the density.

The color palette is defined as following. Consider a density value x, which is in the range between [0,1], then the red intensity value is computed as following:

$$R(x) = \begin{cases} 0, & if\ x \leq \frac{1}{2} \\ 255, & if\ x \geq \frac{3}{4} \\ \left\lfloor \left(\frac{x - \frac{1}{2}}{\frac{1}{4}} \right) * 255 + 0.5 \right\rfloor, & else \end{cases}$$

The green intensity value is computed as following:

$$G(x) = \begin{cases} 255, & if\ \frac{1}{4} \leq x \leq \frac{3}{4} \\ \left\lfloor \frac{x}{\frac{1}{4}} * 255 + 0.5 \right\rfloor, & if\ x < \frac{1}{4} \\ \left\lfloor \left(1 - \frac{x - \frac{3}{4}}{\frac{1}{4}} \right) * 255 + 0.5 \right\rfloor, & else \end{cases}$$

And the blue intensity value is computed as following:

$$B(x) = \begin{cases} 255, & if x \leq \frac{1}{4} \\ 0, & if\ x \geq \frac{1}{2} \\ \left\lfloor \left(1 - \frac{x - \frac{1}{4}}{\frac{1}{4}} \right) * 255 + 0.5 \right\rfloor, & else \end{cases}$$

Let us consider an example. If we have an x value of 0.4 the color intensities are as following:

$$R(x) = 0$$

$$G(x) = 255$$

$$B(x) = \left\lfloor \left(1 - \frac{0.4 - \frac{1}{4}}{\frac{1}{4}} \right) * 255 + 0.5 \right\rfloor = 102$$

Because until now we only calculated the density of the points on the grid, we need to interpolate the values to get a smooth density map. The HTML5 'canvas' element has a method that allows us to do this in a very simple way. We first create a 'canvas' element with the size of the grid and color this according to the density values. Afterwards a new and by a scale factor larger 'canvas' element is created. And finally the old element is simply scaled up to fit the new element. Example Code 5.12 shows how this is implemented.

```
 1: function red(density)
 2: {
 3:    if (density <= 0.5) return 0;
 4:    if (density >= 0.75) return 255;
 5:    return Math.round((density-0.5)/0.25*255);
 6: }
 7: function green(density)
 8: {
 9:    if (density >= 0.25 && density <= 0.75) return 255;
10:    if (density < 0.25) return Math.round(density/0.25*255);
11:    return Math.round((1-((density-0.75)/0.25))*255);
12: }
13: function blue(density)
14: {
15:    if (density <= 0.25) return 255;
16:    if (density >= 0.5) return 0;
17:    return Math.round((1-((density-0.25)/0.25))*255);
18: }
19: var opacity = 180; var scale = 10;
20:
21: var canvas_colors = document.createElement( 'canvas' );
22: canvas_colors.width = width;
23: canvas_colors.height = height;
24:
25: var context_colors = canvas_colors.getContext( '2d' );
26: context_colors.fillStyle = 'rgba(0,0,0,1)';
27: context_colors.fillRect( 0, 0, width, height );
28: var image_colors = context_colors.getImageData( 0, 0, width, height );
29: var data = image_colors.data;
30:
31: for (var i = 0; i < values.length; i++)
32: {
33:    var density = (values[i]-min) / (max-min);
34:    data[i*4] = red(density);
35:    data[i*4+1] = green(density);
36:    data[i*4+2] = blue(density);
37:    data[i*4+3] = opacity;
38:    if (values[i] == 0) data[i*4+3] = 0;
39: }
40:
41: context_colors.putImageData( image_colors, 0, 0 );
42:
43: var canvas = document.createElement("canvas");
44: canvas.width = scale*width;
45: canvas.height = scale*height;
46:
47: var context_canvas = canvas.getContext( '2d' );
48: context_canvas.scale( scale, scale );
49: context_canvas.drawImage( canvas_colors, 0,0);
```

Example Code 5.12

On lines 1 to 18 of the Example Code 5.12 the three functions that compute the color intensity for the colors red, green and blue are listed. They are implemented as described in the previous paragraphs. As first thing, on lines 22 to 24, a 'canvas' element of the size of the grid is created. Afterwards, on line 26 the getContext method of the 'canvas' element is called. This method returns an object that provides methods to draw on the 'canvas' element. Then the whole canvas is filled with a rectangle of black color. The array that contains the color data of the 'canvas' element is on line 29 stored in the data variable. The coloring itself is done on lines 31 to 39. There we iterate through all density values that we received from the PHP script. On line 33 the density value of every grid point is normalized to be in the interval [0,1]. And then the color data of the 'canvas' element are overwritten with the calculated color intensities that correspond to the given density values. On lines 37 and 38 the opacity of the pixel is set to the previously defined 180 if some density value is available and to 0 otherwise. When all pixels of the 'canvas' element are colored the new calculated image data is drawn on the 'canvas' element on line 41. Afterwards on lines 43 to 45 a new 'canvas' element, which is with the scale factor 10 greater than the other one is created. Finally on line 48 the scale method of the context of the 'canvas' element is called. This method scales up everything that is put onto the 'canvas' element afterwards with bilinear interpolation. So at the very end we draw the image of the old 'canvas' element on the new element and the image is ready to be used.

Placing the image on the map

As last step, the image respectively the 'canvas' element, that displays the densities, needs to be placed on the map. This is done using an OverlayView of the Google Maps API. With the OverlayView one can put Objects, which are tied to latitude/longitude coordinates, on the map, like that the Objects move when dragging or zooming the map. Example Code 5.13 shows how this step is implemented.

```
 1: function DensityOverlay(map){
 2:    this.div = null;
 3:    this.bounds = null;
 4:    this.setMap(map);
 5: }
 6:
 7: DensityOverlay.prototype = new google.maps.OverlayView();
 8:
 9: DensityOverlay.prototype.onAdd = function() { }
10: DensityOverlay.prototype.onRemove = function() { }
11: DensityOverlay.prototype.draw = function()
12: {
13:    var overlayProjection = this.getProjection(),
14:    var sw = overlayProjection.fromLatLngToDivPixel(this.bounds.getSouthWest());
15:    var ne = overlayProjection.fromLatLngToDivPixel(this.bounds.getNorthEast());
16:
17:    var div = this.div;
18:    div.style.left = sw.x + 'px';
19:    div.style.top = ne.y + 'px';
20:    div.style.width = (ne.x - sw.x) + 'px';
21:    div.style.height = (sw.y - ne.y) + 'px';
22: }
23:
24: DensityOverlay.prototype.update = function(data)
25: {
26:    //Steps 1-5 are executed here
27:
28:    this.div = document.createElement('DIV');
29:    this.div.style.position = "absolute";
30:    this.div.appendChild(canvas);
31:
32:    var panes = this.getPanes();
33:    panes.overlayLayer.appendChild(this.div);
34: }
```

Example Code 5.13

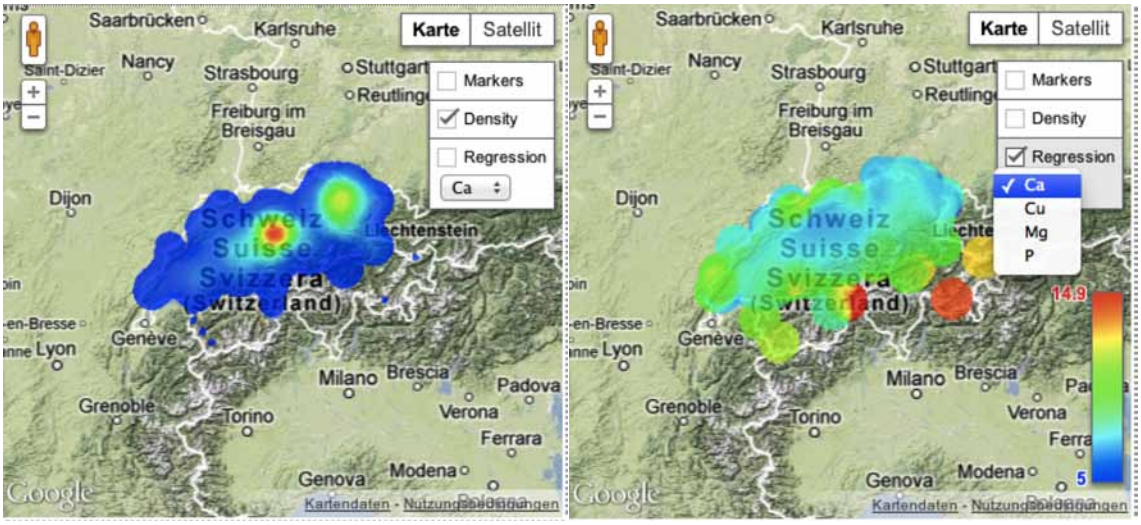
On lines 1 to 5 the constructor of the DensityOverlay class is defined. It takes the Google Map, on which the density image needs to be placed, as a parameter and sets it to its map on line 4. The class has two new member variables, which are as first the 'div' element, which at the end contains the image and the bounds, which are set when the data from the PHP script arrive and contain the information about where to put the image on the map. On line 7 the DensityOverlay is instantiated as a new subclass of the Google Maps API Overlay View class. This class has the methods onAdd, onRemove and draw as functions. The draw function is the most important for this implementation. It gets called when the user zooms in or out of the map. Because the image needs to change its position and size when the user zooms in or out, we handle this resizing and repositioning there. On line 13 the projection of the map is stored. This projection allows us to get the new position of the previously stored bounds on the map. With this information we can resize the 'div' element, which

contains the density image. As last method of the DensityOverlay, the update method is implemented. This method is called when a new density image must be calculated, basically when the user makes a new selection. In this method the five previously described steps of the implementation are called. After all this steps were executed a ‘div’ element is created on line 28. On line 30 the ‘canvas’ element is appended as child to the ‘div’ element. And finally on line 33 the ‘div’ element is put into the pane of the Google Map.

6 Evaluation

6.1 Functional

The interface to use the new available visualization methods is described in this section. In addition some information that can be obtained by using the visualization methods are presented. Figure 6.1 shows the online interface for a selection of the hay feed samples and the nutrients calcium, cooper, magnesium and phosphor.



(a) density visualization. (b) calcium containment visualization.

Figure 6.1: Online interface.

On the map of the online interface some new control buttons were added. These allow the user to display the density image respectively the Kernel Regression for a selected nutrient. The density image is always calculated for all selected data samples therefore no selection of the nutrient is necessary. But the Kernel Regression is always for only one nutrient. Thus a dropdown box is available. With this the users can choose for which nutrient the Kernel Regression should be displayed. This box can be seen in Figure 6.1 (b).

For the Kernel Regression also a scale is shown. This scale allows the users to see for which values of the nutrient the colors are. For example in in Figure 6.1 (b) the red color means, that the hay samples from the red colored region have a calcium containment of about 14.9 and the regions that are blue a containment of about 5.

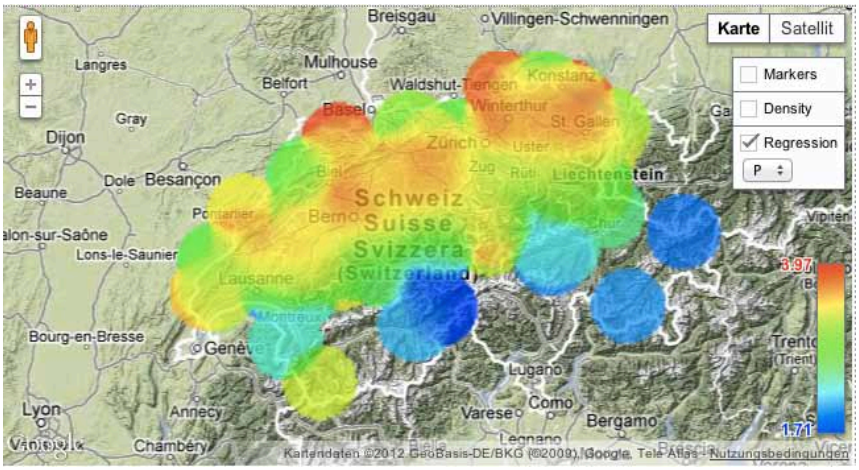


Figure 6.2: Phosphor containment of hay.

If we have a look at the phosphor containment of hay shown in Figure 6.2, we can see, that the resulting image has quite smooth changes from one color to the other. Especially at locations where the density of data sample is high. What we clearly can see from this visualization is, that the phosphor containment of the hay samples is much less in the mountain region than in the lower middle land. To visualize such differences is the main aim of this implementation.

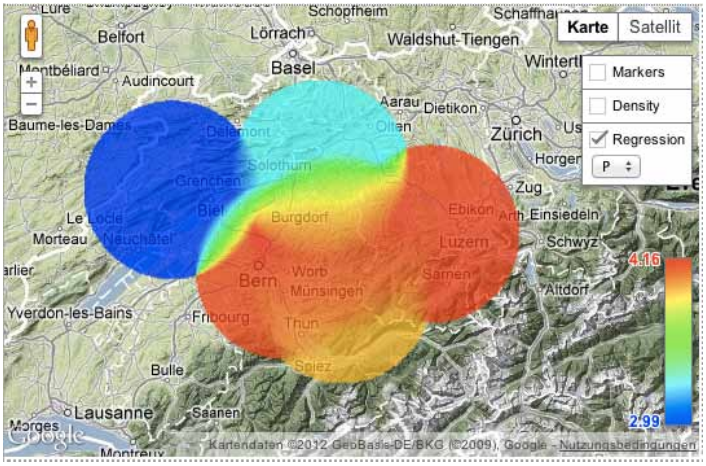


Figure 6.3: Example of Kernel Regression with only few data samples.

The used regression technique has not only advantages. A disadvantage appears when only few data samples are available, as shown in Figure 6.3. The Kernel Regression then has the tendency to show large circles of the same color. This is because in the Kernel Regression a single data sample has an impact on the surrounding region and the size of such a region is determined by the value of h_{opt}. Since h_{opt} is high for a small number of data points, the regions of the influence, i.e., the circles in the image, become large.

6.3 Experimental

In this section an experimental evaluation of the implemented algorithms is presented. We compare the Greedy Approach to the Runtime Efficient Approach and we show that the running time is no longer dependent on the number of data points that are returned by the query or the resolution of the density image. All evaluations were done with the same settings. A computer running Mac OS X 10.8 with 4 GB ram and a 2.4 GHz Intel Core 2 Duo processor was used. The execution was done using the Chrome browser, except for the tests where the running time on the different browsers was evaluated. To test the performance in the Internet Explorer browser a similar computer running Windows 7 was used. The newest versions of all browsers were used for the evaluation.

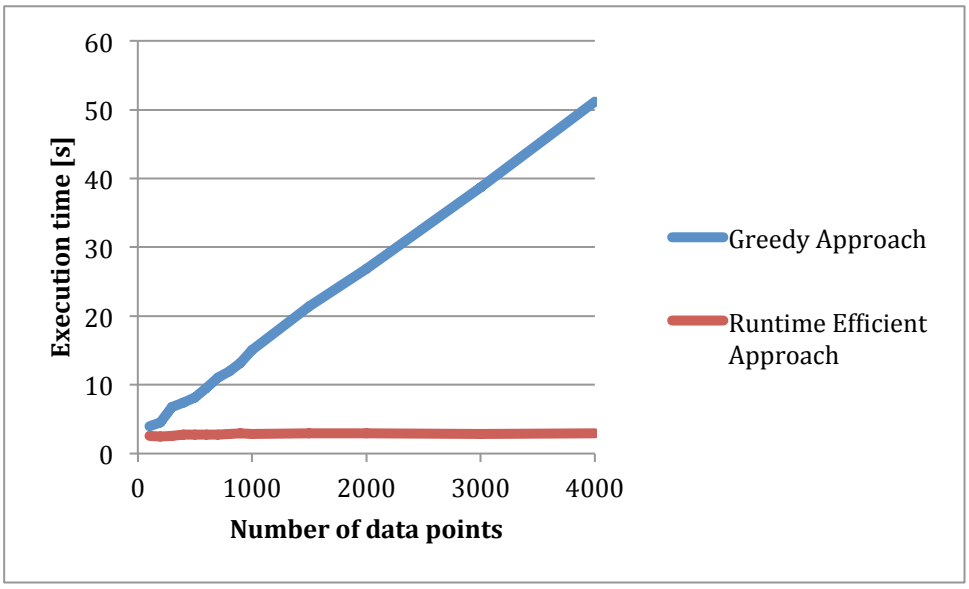


Figure 6.4: Execution time dependent on the number of data points.

Figure 6.4 shows the duration of the calculation for the Greedy Approach and the Runtime Efficient Approach dependent on the number of data points that are retrieved by the query. It can be seen that the running time of the Greedy Approach linearly grows when the number of data points increases. Whereas the Runtime Efficient Approach runs nearly the same time independent of how many data points are considered. This can be explained with the fact that the number of locations is the crucial factor for the Runtime Efficient Approach and not the number of data points itself.

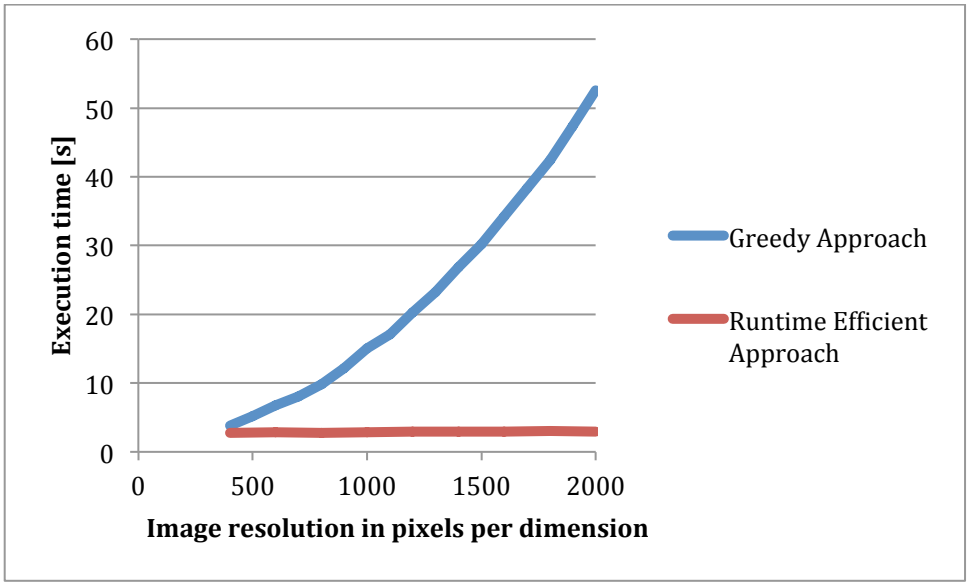


Figure 6.5: Execution time dependent on the used image resolution.

According Figure 6.5 the execution time for the Runtime Efficient Approach is not dependent on the image resolution. This can be explained with the fact that the density values are calculated for the grid points of the sparse grid. Whereas for the Greedy Approach the running time grows quadratically with the increase of the image resolution. This is because the algorithm calculates the density value of every pixel of the image.

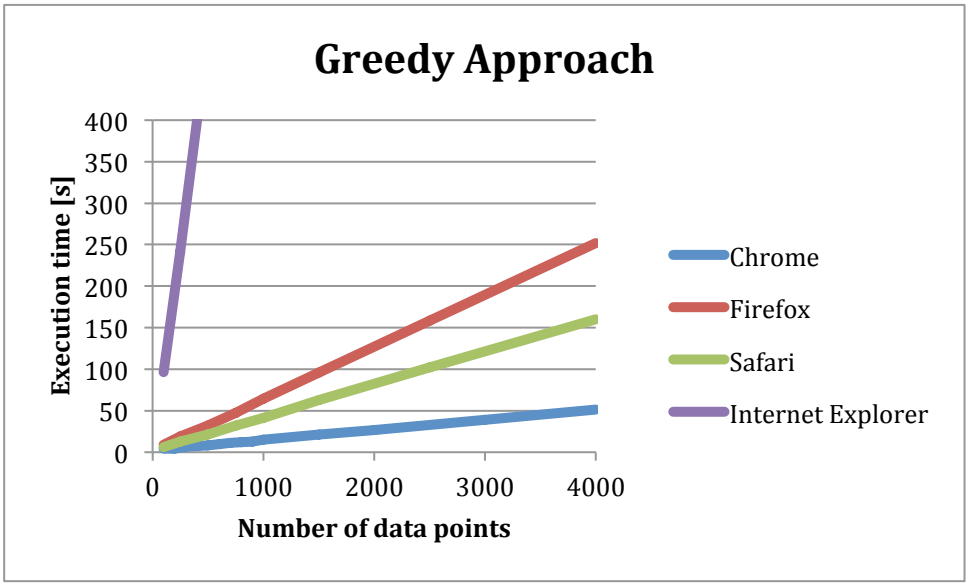


Figure 6.6: Execution time of the Greedy Approach in four different browsers

Figure 6.6 shows that the performance of the Greedy Approach is strongly dependent on the browser that is used. It can be seen that the Chrome browser is way faster than the other browsers. The Safari and Firefox browsers already need a minute to compute the density for 1000 data points. By far the slowest browser is the Internet Explorer. The calculation of the density image for only

100 data points already takes more than a minute. What also can be seen is that the execution time grows linearly to the number of data points for all browsers.

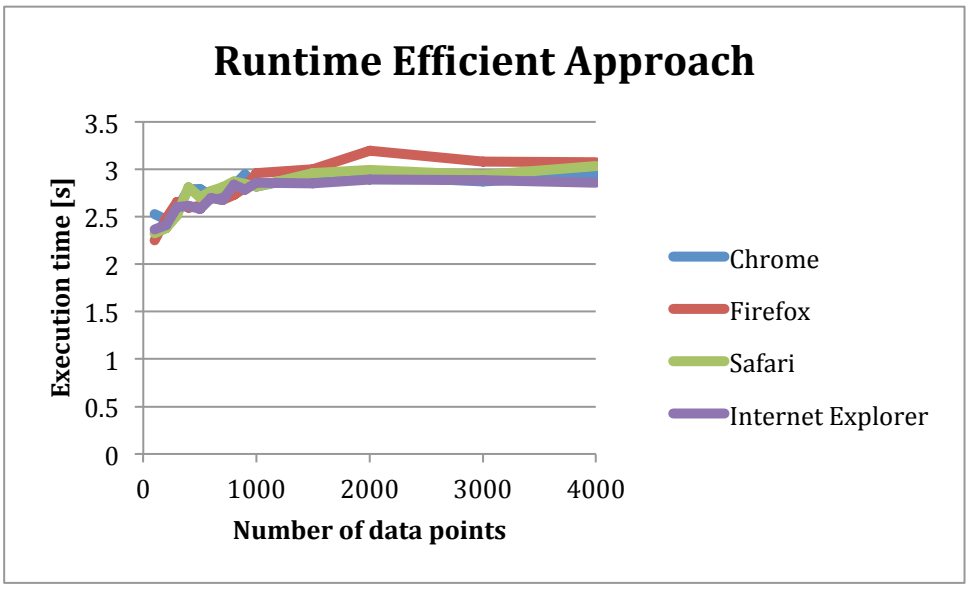


Figure 6.7: Execution time of the Runtime Efficient Approach in four different browsers.

According to Figure 6.7 the execution time of the Runtime Efficient Approach is nearly the same in all four browsers. This shows that the algorithm is usable no matter what browsers the users have. Whereas the JavaScript implementation as the Greedy Approach have the disadvantage, that they are very dependent on the users browser choice, as we have seen in Figure 6.6.

7 Conclusion and Future Work

The high complexity of the Kernel methods makes the visualization of the density a challenging task. However, despite this we managed to develop an algorithm, which computes the density images in an efficient manner. With the introduction of a sparse grid we were able to reduce the running time of the algorithms from quadratic to linear, and with bilinear interpolation we ensure that the resolution of the resulting density image is still high enough. At the end of this work the visualization of the density of the feed samples and the visualization of nutrient containment of feed samples are new functionalities of the online application of the Swiss Feed Database.

The evaluation part of this thesis clearly evidences that the developed algorithms execute independently of the users' browsers and selection criteria in a pleasing time. Moreover the new functionalities allow the users to visually compare the density of feed samples and the nutrient quality of feed samples from different regions with each other. Especially for regions where a lot of data samples come from the algorithm delivers useful information.

Although the visualization delivers good information it also has some restrictions. First of all, the visualization allows only limited comparison of two different selections for the same nutrient. This is because the bandwidth that is used for the Kernel methods changes and because the different colors not always represent the same nutrient value. As second, a similar but a bit expanded problem is, that the change of the nutrient containment over the years cannot be visually seen. The challenges might be to figure out how the image changes when some new measurements need to be considered and when to replace old measurements by new ones. It is left open to find a solution to address these problems in future works.

8 References

[1] https://developers.google.com/maps/documentation/javascript/ (15.08.2012)

[2] http://www.agroscope.admin.ch/ (15.08.2012)

[3] http://www.w3.org/DOM/ (15.08.2012)

[4] http://www.w3.org/TR/XMLHttpRequest2/ (15.08.2012)

[5] Härdle, Müller, Sperlich, Werwarz (1995). *Nonparametric and Semiparametric Models.* Springer, Berlin.

[6] Scott, D.W. (1992). Multivariate Density Estimation: Theory, Practice, and Visualization. John Wiley & Sons, New York.

[7] Silverman, B.W. (1986). Density Estimation for Statistics and Data Analysis. Chapman and Hall, London.