Computational Science and Engineering
(International Master's Program)

Technische Universität München

Master's Thesis

Integrated approach of Random Projections and Sparse Grids for Density estimation

Oriolson Rodriguez Ramirez

Computational Science and Engineering (International Master's Program)

Technische Universität München

Master's Thesis

Integrated approach of Random Projections and Sparse Grids for Density estimation

Author:	Oriolson Rodriguez Ramirez
1st examiner:	Univ.-Prof. Dr. rer. nat. habil. Hans-Joachim Bungartz
Assistant advisor:	M.Sc. Severin Maximilian Reiz
Submission Date:	November 15st, 2020

I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.

November 15st, 2020 Oriolson Rodriguez Ramirez

Acknowledgments

First, I would thank M.Sc. Severin Reiz for sharing all his invaluable knowledge and being so patience during the development of this challenging topic. His insightful feedback brought this thesis to a higher level.

I also would like to thank the Scientific Computing chair for giving me the opportunity to take part in this great master program.

In addition, I would like to thank my family: Doña Rosa, Tio miro, Abe, Faby, Rosendo and Calaca, and of course, all my nieces and nephews, for their wise counsel and support during the hard times.

I could not have completed this project without the support of my friends, Dani, Hisham, Brenda, Peter, Didier, Jonathan, and Samuel, who provided happy distractions to rest my mind outside my research.

The idea of a learning machine may appear paradoxical to some readers. How can the rules of operation of the machine change? They should describe completely how the machine will react whatever its history might be, whatever changes it might undergo. The rules are thus quite time-invariant. This is quite true. The explanation of the paradox is that the rules which get changed in the learning process are of a rather less pretentious kind, claiming only an ephemeral validity (...)
An important feature of a learning machine is that its teacher will often be very largely ignorant of quite what is going on inside, although he may still be able to some extent to predict his pupil's behavior.

-Alan Turing

Abstract

Sparse Grid Density Estimation faces two challenges when it is applied to extremely high-dimensional and clustered data. The first is the *Curse of dimensionalty* which makes the required computing power increase exponentially with the number of dimensions. Despite that Sparse Grids moderate the effect of the dimensionality it can reach a point in which a computation is unfeasible. The second challenge is the location of some grid points; at the beginning of the computation the grid spreads all over the domain, including places where there is no data at all. When clustered data is used all samples are located around cluster centroids and not scattered all over the domain, this makes the phenomenon of placing grid points where they are not really necessary, to accentuate. This unwanted product adds computational cost but does not improve accuracy. Despite that Adaptive Refinement have shown very good results, the computation will carry all the way those inefficient grid points added at the beginning.

To reduce the undesired aforementioned effects a pipeline is implemented. This pipeline is divided in two parts: The first one is a Locality Sensitive Hashing algorithm that uses Random Projections to divide the data in subsets that correspond to clusters. The second part applies dimensionality reduction to embed the whole data set and each cluster to a significantly lower dimensional space and then compute density estimations using Sparse Grids. For both parts two different sets of metrics are introduced to measure the efficacy of this implementation. Finally, the *Conventional Approach* using the whole data set is compared to two different methodologies to treat each individual cluster in parallel(*Cluster Analysis* and *Clusters Extraction*).

Results for a real-world data set and Synthetic one show that the Conventional Approach presents better results than Cluster Analysis in most of the scenarios. However, Cluster Extraction exhibit very good results compare to those two methodologies in 60% to 80% of the proposed numerical experiments. This promising results are consistent even in scenarios when the numerical experiments have very similar computational cost.

Contents

Acknowledgements vii

Abstract ix

I. Introduction and Background Theory 1

1. Introduction 3

2. Theory 5
2.1. Density Estimation . . . 5
2.1.1. Parametric Density Estimation . . . 5
2.1.2. Semi-parametric Density Estimation . . . 7
2.1.3. Non-Parametric Density Estimation . . . 10
2.2. Sparse Grids for Density Estimation . . . 11
2.2.1. Sparse Grids . . . 12
2.2.2. Density Estimation Using Sparse Grids . . . 12
2.3. Dimensionality Reduction . . . 13
2.3.1. Principal Component Analysis . . . 14
2.3.2. Random Projections . . . 15
2.4. Locality Sensitive Hashing . . . 17
2.4.1. l_p-LSH Family . . . 18
2.4.2. Distance Metrics in High Dimensional Spaces . . . 19

II. Implementation 21

3. Pipeline 23
3.1. Random Projection . . . 23
3.1.1. Finding the Best Possible Projection . . . 26
3.2. Group Sorting . . . 26
3.2.1. Distance Metrics, Sketch Construction and Parameter Learning . . . 28
3.2.2. LSH Algorithm: Putting All Pieces Together . . . 30
3.2.3. Validation Metrics for LSH . . . 34
3.3. Dimensionality Reduction of Clusters . . . 34

3.4. SG++ Library . . . 35
3.4.1. General Settings . . . 37
3.4.2. Grid Configuration . . . 38
3.4.3. Validation Metrics for SGDE . . . 38

4. Numerical Experiments **43**
4.1. Input Pipeline Parameters: Summary . . . 43
4.2. Data Sets . . . 43
4.3. Numerical Experiments . . . 44
4.3.1. LSH Numerical Experiments . . . 45
4.3.2. Dimensionality Reduction and SGDE Numerical Experiments . . . 45

III. Results and Conclusion **49**

5. Results and Analysis **51**
5.1. LSH Results for Gene Expression Data Set . . . 51
5.2. LSH Results for Synthetic Data Set . . . 53
5.3. Dimensionality Reduction and SGDE Results for Gene Expression Data Set 53
5.4. Dimensionality Reduction and SGDE Results for Synthetic Data Set . . . 55

6. Conclusions **59**

Bibliography **59**

Part I.

Introduction and Background Theory

1. Introduction

The *Curse of Dimensionality* is a term introduced by Richard E. Bellman [7] [8]. It shows how by increasing the number of dimensions in a given space, it also expands its volume exponentially, adding sparsity to the data contained in that space. This makes imperative to increase the amount of data exponentially in order to make statistical results drawn from that data to be reliable. This phenomenon affects directly machine learning algorithms that uses high dimensional data sets, these methods also require exponential growth of computing power to give results in a reasonable amount of time.

Sparse Grids are used for density estimation, which is also affected by the Curse of Dimensionality, however, this method reduces some of the undesired effects that a highly dimensional data set brings with it. Pflüger [27], Bungartz et al [10], Peherstorfer [25] and others, using SG++ library [33], have proven how this methodology can give very good results for this kind of problems. Despite that there is still space for some improvements, one of those is the way how the Sparse Grid is distributed in the domain. Usually, the grid points are placed in the whole domain of the input space, including some areas where only a few are needed or none at all. This grid points add computational cost but does not improve accuracy to the results [23]. Despite that Adaptive Refinement can be used, it still can be applied more efficiently if grid points are distributed more efficiently in an scenario of clustered data.

The main objective of this thesis is to test if by subdividing the initial data set into clusters it can improve the accuracy of Sparse Grid Density Estimation. For this purpose, I use customized grids that only converts the domain where the actual samples are. This tries to avoid the use of grid points in areas where there is no samples.

In this thesis I implement a pipeline that 'preprocess' a highly dimensional clustered data set using already existing Machine Leaning and Data mining algorithms in order to boost accuracy of Sparse Grid Density Estimation (SGDE). This pipeline consist of two parts; the first one applies a *Random Projection* to reduce dimensionality then a *Locality Sensitive Hashing* (LSH) algorithms to hash all similar instances to same buckets. Theses buckets are considered clusters that later I use in two different methodologies, one I call *Clusters Analysis* and the other *Cluster Extraction*. The second part of the pipeline is compounded by Dimensionality reduction and SGDE. The former is done either with *Principal Component Analysis* (PCA) or Random Projection then PCA. And the later is the actual SGDE computation with customized grids.

All inputs to SGDE are previously reduced to lower dimensional space to make this computations feasible in a reasonable time. What changes is the input data which can be either the Whole Data Set when executing the problem with the *Conventional Approach* or one cluster projected to a lower dimensional space then input to SGDE (Cluster Analysis) or one cluster that is taken from the projected whole data set (Cluster extraction), then input to SGDE. The two cluster methodologies are applied in a parallel computation to all of them in the data set. In other words, the difference between the two cluster's methodologies is where that cluster is taken from, either form already projected data (Cluster Extraction) or it is taken from the input space and then projected (Cluster Analysis). In all the methodologies (except in the Conventional approach) a customized grid with *Bounding Boxes* is used for SGDE.

The first chapter that you find is the Theory and Background in which I show all the theoretical basis for Density Estimation, Sparse Grid Density Estimation, Dimensionality Reduction and Locality Sensitive Hashing. I make emphasis on the methods that are used latter in the pipeline. Second comes the Implementation Chapter that contains all the details of the pipeline: order of execution, libraries that are used, code snippets and pseudo-code for the algorithm that were coded. Third, I show all scenarios that are proposed to test the different approaches and the data sets that are used. This is called the Numerical Experiment chapter. Finally, Result and Conclusion chapter is presented with the comparisons between all experiments done for each data set. Then I expose all the conclusions that are drawn from all this process.

2. Theory

In this chapter an overview of the theoretical background and techniques used for the realization of this thesis are presented. Starting with Density Estimation as a general case. Showing the main methods to find the Probability Density Function (PDF). Then, I introduce density estimation using Sparse Grids in the second part of the chapter. In the third section, a brief review of some dimensionality reduction algorithms is given. With focus on Random Projections (dense and sparse) and the core of this methodologies: The Johnson-Lindenstrauss Lemma. Finally, the theory behind Locality Sensitive Hashing is presented. Because the main focus is put on high dimensional spaces and due to an unusual behavior of vectors distances in this spaces, a short description of this behavior and its possible approaches are shown.

2.1. Density Estimation

The aim of Density estimation techniques is to approximate the Probability Density Function (PDF) of an input data [36]. It can either be *parametric*, *semi-parametric* or *non-parametric* depending on the initial assumption of the underlying data . The non-parametric methods make no assumption at all about the underlying data and the parametric algorithms effectively make apriori assumptions about the training set's distribution. The semi-parametric algorithms are a hybrid of these two [36].

Density Estimation methods are categorized as *unsupervised learning* because there is only input data. The objective is to find patterns in the input space without having the correct values $y \in Y$ in the mapping $X \to Y$ [5].

2.1.1. Parametric Density Estimation

The sample set $X := \{x_1, ..., x_n\} \subset \mathbb{R}^d$ is assumed to be drawn from known model (e.g. Gaussian). Based on this assumption a finite number of parameters are computed. Once these parameters are determined, the distribution $p(x)$ is estimated.
This methods can be subdivided according to the dimension of the sample space; When $d = 1$ the method of preference is the Maximum Likelihood Estimation (MLE). On the other hand, when $d > 1$ a generalized case to multivariate is the methods of choice [5].

Maximum Likelihood Estimation

Lets define a set of parameters θ that parameterize the known distribution $p(x,\theta) \forall x \in X$ and that all samples x are independent and and identically distributed (iid). An example of a parameterized distribution is the Gaussian which is parameterized by the mean(μ) and the variance (σ^2). i.e. $x_i \sim \mathcal{N}(\mu, \sigma^2)$.

The objective is to find θ that makes x as likely as possible to $p(x \mid \theta)$. Therefore, the likelihood of θ given the samples space X is defined as [5]:

$$l(\theta \mid X) := p(X \mid \theta) = \prod_{i=1}^{n} p(x_i \mid \theta) \tag{2.1}$$

Where, n is the total number of samples x in the input space X.

Because, the objective is to find θ that maximizes the likelihood, the logarithmic function is applied to equation 2.1. This can be done due to the fact that $log(\cdot)$ monotonically increases, therefore does not change the maximum value.

$$L(\theta \mid X) = log(l(\theta \mid X)) = \sum_{i=1}^{n} log(p(x_i \mid \theta)) \tag{2.2}$$

Finally, maximizing the $log - likelihood$ equation 2.2:

$$\hat{\theta} = \underset{\theta}{\operatorname{argmax}}(L(\theta \mid X)) \tag{2.3}$$

Where, $\hat{\theta}$ is the approximated parameter set of θ. The higher the number of sample points, the better is the approximation of θ ($\lim_{n \to \infty} \hat{\theta} = \theta$).

Multivariate Density Estimation

This is a generalization of univariate Density Estimation shown above. In this case the input is multidimensional ($d > 1$), for that reason the samples are vectors with more than one attribute. The data set can be represented as a *data matrix*.

$$X = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1d} \\ \vdots & \ddots & & \vdots \\ x_{N1} & & & x_{Nd} \end{bmatrix}$$

The columns vectors of the input space are called *attributes* or *features* and they range from 2 to d. The row vectors are usually referred to as *observations* or *instances*. In this thesis the sample X is going to be multidimensional unless stated otherwise.

The main objective is also to find Θ (capital θ) which is a now a set of vectors. This parameter is analogous to the 1 dimensional case. Now, lets consider the Gaussian multivariate distribution which is the most commonly used, the *mean vector* and *covariance matrix* is defined as follows.

$$\boldsymbol{\mu} = [\mu_1 ... \mu_d]^T \tag{2.4}$$

The variance for the 1-dimensional case (one attribute) is σ^2 and the covariance for two attributes x_i and x_j is as follows.

$$\sigma_{i,j} \equiv Cov(x_i, x_j) = E[(x_i - \mu_i)(x_j - \mu_j)] = E[x_i x_j] - \mu_i \mu_j \tag{2.5}$$

Where, E is the expected value of its argument and $\sigma_{i,j} = \sigma_{j,i}$. Based on these the covariance matrix is defined as Σ, which is usually a positive definite matrix [5].

$$\Sigma = \begin{bmatrix} \sigma_1^2 & \sigma_{12} & \cdots & \sigma_{1d} \\ \sigma_{21} & \sigma_2^2 & \cdots & \sigma_{2d} \\ \vdots & \ddots & & \vdots \\ \sigma_{d1} & \sigma_{d2} & & \sigma_d^2 \end{bmatrix}$$

Finally, the PDF for a sample when is assume to be normal in the multivariate case ($X \sim \mathcal{N}_d(\boldsymbol{\mu}, \Sigma)$) is as follows. In this case the argument Θ is omitted.

$$p(X) = \frac{1}{(2\pi)^{d/2}|\Sigma|^{1/2}} \exp[-\frac{1}{2}(X - \boldsymbol{\mu})^T \Sigma^{-1} (X - \boldsymbol{\mu})] \tag{2.6}$$

2.1.2. Semi-parametric Density Estimation

Assuming beforehand the sample distribution reduces the calculation to an small number of parameters, however it could increase the error due to the bias added by the assumed distribution. For example, in case of *clustered data* with more that one cluster (also called groups or components), the normal distribution assumption is violated [5]. In cases like these, the semi-parametric density estimation comes into play, giving more freedom to make some underlying assumptions of each cluster in the input data. There is still an assumption of the parametric model but of each individual group.

Let's define the *mixture density* of X as the weighted sum of different densities 2.7. With k is the number of groups, $p(X|C_i)$ is the cluster density and $P(C_i)$ is the mixture proportions i.e. weights. If $k = 1$ then $P(C_i) = 1$ and it becomes 2.1.

$$p(X) = \sum_{i=1}^{k} p(X|C_i) P(C_i) \tag{2.7}$$

The parameter k is given and the learning process consist of the estimation of the component densities and the weights. Here, the parameter set $\Theta = \{P(C_i), \boldsymbol{\mu_i}, \Sigma_i\}, i = 1...k$.

In this section only a brief presentation of the most common algorithms is given. For a detailed review of this kind of unsupervised learning algorithms refer to Xu et al. [38] [37], Richards J.A. [29] and Greenlaw et al. [16].

Expectation-Maximization Algorithm

The Expectation-Maximization Algorithm (EM) try to approximate the parameters that maximize the likelihood of the sample [13]. Therefore, applying the log function to equation 2.7 gives

$$L(\Theta|X) = \sum_j log \sum_{i=1}^{k} p(x_j|C_i)P(C_j) \tag{2.8}$$

Equation 2.8 is solved iteratively and it is a generalization of the popular *K-Means Clustering* [5]. Where, EM is a more probabilistic algorithm. Redner et al [28] introduced a second set of hidden variables, Z, in case the approximation of $L(\Theta|X)$ is not possible. In that scenario, the *complete likelihood* $L_c(\Theta|X, Z)$ is computed. The two-steps iterative algorithm is as follows:

$$\begin{aligned} E - step &: \mathcal{L}(\Theta|\Theta^l) = E[L_c(\Theta|X, Z)|X, \Theta^l] \\ M - step &: \Theta^{l+1} = \underset{\Theta}{\operatorname{argmax}}\, \mathcal{L}(\Theta|\Theta^l) \end{aligned} \tag{2.9}$$

In equation 2.9, l is the iteration index, $E - step$ is the Expectation step and $M - step$ is the Maximization step. One important thing of this algorithm is the used of $\mathcal{L}$(expectation given X) since Z is a hidden variable the L_c cannot be treated directly.

Hierarchical Clustering

This methodology uses a similarity measure between instances of the sample. Considering that instances in the same cluster are more similar than the instances in other groups, this is exploited to cluster the data. In this algorithm a distance metrics is used; Usually the Euclidean distance is the metric of choice, ensuring that all attributes have the same scale [5]. However, there is a important aspect to highlight about the behavior of the distance metric in high dimensional spaces, where the euclidean distance is not the best choice [3]. To perform the actual clustering an *agglomeration algorithm* is used. This starts with $k = N$, in other words, each instance is a cluster. Then, it starts merging all groups that fall within a given threshold. The strategies considered to merge the neighboring algorithms are: *Single-link clustering* which takes the minimum distance between the two groups (see

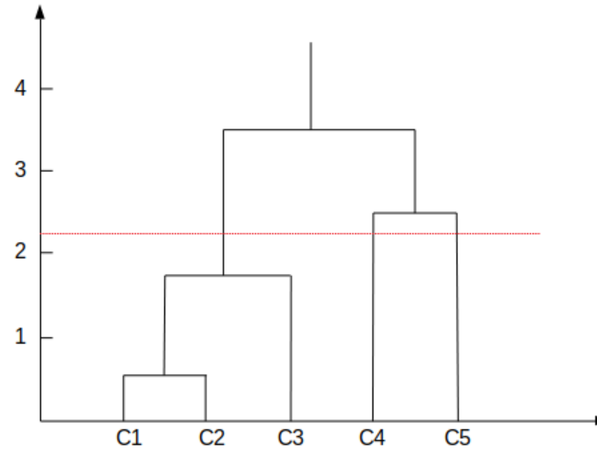


Figure 2.1.: Dendogram Example. Red-dashed line correspond to the given threshold for agglomeration. It starts with five clusters and is agglomerated to 3 three clusters

Equation 2.10) under consideration and evaluates the given threshold. If the similarity is smaller than the threshold, this groups are merged.

$$d(C_i, C_j) = \min_{\boldsymbol{x_i} \in C_i, \boldsymbol{x_j} \in C_j} d(\boldsymbol{x_i}, \boldsymbol{x_j}) \tag{2.10}$$

Complete-link clustering considers the largest distance (see Equation 2.11) between all possible combinations of instances from the two groups under comparison and evaluates the threshold, analogously to the Single-link clustering.

$$d(C_i, C_j) = \max_{\boldsymbol{x_i} \in C_i, \boldsymbol{x_j} \in C_j} d(\boldsymbol{x_i}, \boldsymbol{x_j}) \tag{2.11}$$

There exists other strategies like the *Average-link clustering* and the *Centroids(means)*. Once agglomeration algorithm finishes a useful tool to draw the clusters and their distances between them is the *Dendogram*. This a tree-like diagram where the leaves represent the clusters and edges the sequence in the way they are merged. As an example see Figure 2.1 where there were 5 initial clusters and a distance threshold mark with the horizontal dashed red line. After merging, 3 clusters remain.

The ideas of distance metric for high dimensional spaces and link clustering as agglomeration algorithm gain relevance in the implementation chapter.

Spectral Clustering

The idea behind this algorithm is to not use the original input sample space, but instead one of reduced dimensionality [34]. In a reduced space the difference between the group are more clear therefore a clustering algorithm is more efficient. For that reason a preprocessing step of dimensionality reduction is performed. Then k-means is used in the new space. In the preprocessing step is important to preserve the pairwise similarities and to consider carefully the parameter selection because it could make the result change considerably due to the similarity propagation to neighboring clusters [34].

2.1.3. Non-Parametric Density Estimation

Density estimation using non-parametric methods makes no assumptions regarding the underlying distribution of the input data, except for "similar inputs gives similar outputs" assumption [5]. Therefore, the algorithm complexity depends on the training data set exclusively. Usually, train data sets has more instances than features ($N > d$) which results in the biggest disadvantage of non-parametric algorithms: They involve more computation than parametric methods. But, sometimes it is not the case and there can be more features than instances which could lead to overfitting. This is going to be treated in later sections.

The conventional non-parametric methods require $O(N)$ in memory and $O(N)$ of computational cost, compared to $O(d)$ or $O(d^2)$ of parametric methods. However, there exists more advanced algorithms that have different memory and computation requirements. These techniques often suffer from the *curse of dimensionality*[7][8] for data sets with a high number of features.

Kernel Density Estimator

Kernel Density Estimator (*KDE*) is the most common non-parametric density estimation algorithm for multivariate data. In this case $\boldsymbol{x_i} \in X^{(N)}$ comes from an unknown iid distribution $f(\boldsymbol{x})$, approximated by the $p(\boldsymbol{x})$. This method uses an smooth weight function $K(\cdot)$, *kernel function* which is smooth and inherits this property to $p(\boldsymbol{x})$. The most commonly used kernel is the multivariate Gaussian (see Equation 2.12). One of the most important properties of kernel is that the integral over the whole domain must be equal to one [5].

$$K(\boldsymbol{u}) = (\frac{1}{\sqrt{2\pi}})^d \exp(-\frac{\|\boldsymbol{u}\|^2}{2}) \tag{2.12}$$

The estimated density is defined as follows:

$$p(\boldsymbol{x}) = (\frac{1}{Nh^d}) \sum_{i=i}^{N} K(\frac{\boldsymbol{x} - \boldsymbol{x_i}}{h}) \tag{2.13}$$

Where, parameter h in 2.13 is the length of the interval or bin. Here, h is a parameter to optimize, specially in high dimensional spaces where it can dramatically increase the computational cost and affects accuracy.

Histogram Estimators

This technique is the most popular one in the non-parametric family. It consist of two basic steps; The first is to define fixed consecutive intervals of length h, called bins in each dimension of the samples. Then, as a second step, the algorithm test if the instances in an specific dimension fall in one of the bins. Therefore, a *frequency* is defined for each bin based on the number of samples that falls in it. Contrary to KDE, this algorithm is not continuous with jumps between two bins. [5]

2.2. Sparse Grids for Density Estimation

This algorithms places an *Sparse Grid* on the domain where the input sample lays and uses a set of *Basis Functions* to make a grid point wise approximation of density functions. It bases its approach on Multigrid methodologies, Sparse matrices, Finite Elements theory and Adaptive Mesh refinement to accomplish the task [33]. This method is mainly used when dimensionality reduction algorithms cannot be applied, therefore, conventional machine learning algorithm fail due to the Curse of Dimensionality.
All the formulae and information used in this section is extracted from Bungartz et al [10] and Peherstorfer [25]. Here, I also show the basics of Sparse Grids, to see a more detailed explanation you can refer to those two references mentioned before.

To find the approximated $\hat{f}(x)$ of the unknown function $f(x)$, most of the conventional algorithms iterate over the whole N-samples in the data set ($X = \{x_1, ...x_N\}$). Evidently, when the number of samples increase the computational cost of the algorithm also increases. On the other hand, Sparse grids calculate $\hat{f}(x)$ as a linear combination of weights α_i and the basis functions $\phi_i(x)$ (See Equation 2.14). This make Sparse Grids independent of the number of samples.

$$\hat{f}_n(x) = \sum_{i=1}^{n} \alpha_i \phi_i(x) \tag{2.14}$$

Where, $\phi_i \in \Phi = \{\phi_1, ..., \phi_n\}$, $f_n \in V_n$-space and n is the *grid level*. One example of a linear basis function with compact support is $\phi(x) = max\{1 - |x|, 0\}$. The α_i are the solution of a linear system of equations that we see in later sections.

2.2.1. Sparse Grids

Lets define an Sparse Grid of level n in one dimension with 2^n number of intervals, subdivided equidistantly ($h_{n,i} = ih$) in intervals of $h := 2^{-n}$ length

$$\Omega_n := \{x_{n,i} | i = 0, ..., 2^n\}, \tag{2.15}$$

Therefore the number of grid points depend exponentially on the number of dimensions ($\mathcal{O}(h_n^{-d})$). The subindex n also revels the underlying principle of Sparse Grids that uses a one dimensional Hierarchical system of basis functions and a the tensor product to generalize results to the d-dimensional case. Also for the generalization of grid points the Cartesian product is used.

In the general case to d-dimensions the subindexes i and n become vectors and the basis function is the product of all 1-d basis functions.

$$\begin{aligned} \vec{n} &= (n_1, ..., n_d) \\ \vec{i} &= (i_1, ..., i_d) \\ \phi_{\vec{n},\vec{i}} &= \prod_{k=1}^{d} \phi_{n_k,i_k}(\boldsymbol{x}) \end{aligned} \tag{2.16}$$

This generalization leads in a sparse grid space V_n^1 of level n. Defining I as the set of all level index pairs and rewriting Equation 2.14 for this new grid space

$$\hat{f}_n(x) = \sum_{n,i \in I}^{n} \alpha_{n,i} \phi_{n,i}(\boldsymbol{x}) \tag{2.17}$$

In Equation 2.17 $n = |I|$ and all i and n are vectors.

An important characteristic to notice is that Sparse Grids usually use *Spatial and dimensional refinement* to reduce the number of grid points, hence the computation cost. This algorithm computes the values of $|\alpha_{n,i}\phi_{n,i}|$ and select the highest values. Then, takes the grid points that correspond to those values and refine them by applying a hierarchical method to those grid points (it basically adds more grid points). The number of grid points to refine can be control either by a threshold value of $|\alpha_{n,i}\phi_{n,i}|$ or by selecting the top z-values of that computation (z is a predefined number of grid points to be refined). To read about dimensional refinement refer to Buse [11]

2.2.2. Density Estimation Using Sparse Grids

Sparse Grid Density Estimation (SGDE) is a technique that approximate the PDF using sparse grids. To define Density Estimation using sparse grid we start with the equation ob-

tained when the density is estimated using spline smoothing [18], and applying Galerkin method, we look for $\hat{f}_n \in V_n^1$ such that [25]

$$\int_\Omega \hat{f}_n(\boldsymbol{x}) \cdot \phi(\boldsymbol{x}) d\boldsymbol{x} + \lambda \int_\Omega L\hat{f}_n(\boldsymbol{x}) \cdot L\phi(\boldsymbol{x}) d\boldsymbol{x} = \frac{1}{N} \sum_{i=1}^{N} \phi(\boldsymbol{x_i}) \tag{2.18}$$

It holds $\forall \phi \in \Phi$. λ is the regularization parameter, L is the Laplacian and N is the number of samples.
Replacing Equation 2.17 in 2.18 and taking $R_{i,j} = (\phi_i, \phi_j)_{L^2}$, $C_{i,j} = (L\phi_i, L\phi_j)_{L^2}$ and $b_i = \frac{1}{N}\sum_{j=1}^{N} \phi_i(\boldsymbol{x_j})$

$$(\boldsymbol{R} + \lambda \boldsymbol{C})\boldsymbol{\alpha} = \boldsymbol{b} \tag{2.19}$$

The linear system of equation in 2.19 is solved to find $\boldsymbol{\alpha}$. In the linear system of equations all matrices are of size $(M \times M)$ and the vector b is of size M, where M is the number of grid points. For SGDE matrix $\boldsymbol{C}$ can be either ∇ or the Identity matrix.

2.3. Dimensionality Reduction

When using data sets with many features, i.e. high dimensional data, the implementation of machine learning algorithms becomes challenging due to the computational cost to evaluate the corresponding function in each data point (e.g. similarity search) [17]. Furthermore, high dimensional data can cause troubles in some algorithms when its dimensions are correlated and it is difficult to visualize [5].

Dimensionality reduction makes reference to two groups of algorithms: *Feature Extraction* and *Feature selection*. The former consist of construction an embedding on a space with reduced dimensions than the original. This is done by performing a combination of the original features e.g linear combinations [6]. In this classification we find algorithms like Principal Component Analysis(PCA) and Random Projections (RP), which we are going to analyze in latter sections. And Feature Selection consist of finding the most relevant dimensions, and discarding the remaining features [5]. The most relevant algorithm for this is the *Subset Selection* which looks for the most relevant important features (the ones that contribute more to accuracy) [5]. This type of dimensionality reduction is not going to be shown because it plays no role on this document.

Feature Extraction

Dimensionality reduction takes $x \in \mathbb{R}^d$ to a subspace $x \in \mathbb{R}^R$ where $R << d$, increasing the differences between instances and, hopefully, preserving most of the information [6]. Dimensionality reduction is a sub-field of unsupervised learning algorithms where many

characteristic of the data are compressed either to a subset or to a single label [9]. The label represents the cluster where it belongs to and the subset represents a sub-space of the original d-dimensional space.

A lower dimensional data set can be used as an stand-alone method for visualization or as a preprocessing step before applying clustering or any other algorithm that requires low dimensional data sets. Now lets define a data matrix (data set in matrix representation) as $X \in \mathbb{R}^{Nxd}$, where N represents the number of instances and d number of features of every instance:

$$X = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1d} \\ \vdots & \ddots & & \vdots \\ x_{N1} & & & x_{Nd} \end{bmatrix}$$

The element x_{ij} of X represents the $i-th$ feature of $j-th$ instance.

Usually, data can be projected to a sub-space $\mathbb{R}^R$ (a.k.a lower dimensional manifolds), with minimum loss of information [12]. The processes can be done either by multiplying X by an orthogonal (or ”almost” orthogonal matrix) as it is done in Random Projections or by decomposing X as PCA or Singular Value Decomposition (SVD) does.

2.3.1. Principal Component Analysis

In PCA the coordinate system is transformed such that the variance between the remaining axis is maximized [5]. This is basically a projection of x onto the direction of w, it is mathematically expressed as follows.

$$z_i = w_i^T \cdot x_j \tag{2.20}$$

Where, w_i is called *principal component* of i-th feature and it is required that $\|w_i\| = 1$. By applying equation 2.20, we are amplifying the difference between samples [17]. To maximize the variance of z_i we rewrite equation 2.20 as a Lagrange Problem but previously replacing x by its covariance matrix Σ and multiplying both sides by w_1 form the right. We fixed $i = 1$ to indicate that it is the analysis for the first principal component. This results in equation 2.21. To a more detailed mathematical derivation you can refer to Alpaydin (2014)[5].

$$\max(var(z_1)) = \max_{w_1}(w_1^T \Sigma w_1 - \alpha(w_1^T w_1 - 1)) \tag{2.21}$$

By calculating the derivative with respect to the principal component and setting to zero, we obtain

$$\begin{aligned} w_1^T \Sigma w_1 &= \alpha w_1^T w_1 \\ w_1^T w_1 &= 1 \end{aligned} \tag{2.22}$$

Therefore, $\alpha = \lambda_1$ which is an eigenvalue of Σ and w_1 its eigenvector. And to maximize it we take the w with highest λ. By doing this procedure we are actually calculating the eigenvalues and eigenvectors in each iteration. For $\max(var(z_2))$ a similar process is done. But considering that the calculation of λ_2 and w_2 is totally independent of the computation for λ_1 and w_1.

2.3.2. Random Projections

Algorithms of random projection are also useful for dimensionality reduction, in these cases, matrix X is multiplied by another orthogonal or *almost*-orthogonal random matrix $A \in \mathbb{R}^{d \times R}$. By doing the dot product of a row vector of X and a column vector of A ($\vec{x_i} \cdot \vec{a}$), the input space for i-th instance is reduce from d to R, where ($R << d$). This operation preserves all pairwise distances with an small arbitrary distortion ϵ [1]. One important fact of A is that its elements are totally independent of X, this is a big difference with respect to PCA where X is factorized.

$$X_{RP} = X \cdot A \tag{2.23}$$

Equation 2.23 is basically projecting each row vector of X to a subspace, it leads to a computational cost of $O(dRN)$. In this case the more orthogonal are the column vectors of A with respect to $\vec{x_i}$, the more it preserves the pairwise distances. If A is completely orthogonal, ϵ is equal to zero, therefore, no loss of information. However, it is more expensive to find such a matrix rather than chose the elements of $\vec{a_j}$ randomly. This works because in high dimensional spaces you can find almost any orthogonal direction. Hence, despite the randomness nature of A, the probability that it is almost-orthogonal to X increases [9].

There are two important aspects to control in these cases: the distribution from which the elements of $\vec{a}$ are drawn and the dimension of the embedded space R. The former derives in the different types of Random Projections e.g. Sparse, Gaussian etc. And the latter can also be exchange for ϵ. In case R is given, ϵ becomes an observed parameters of the added distortion, the contrary happens when ϵ is the input parameter.

Johnson–Lindenstrauss Lemma

The Johnson and Lindenstrauss (JL) lemma [20] is a fundamental result that allows the construction of a proper matrix A with optimal R dimensions (See Equation 2.24). It shows how a random transformations preserves the inner product of the vector plus $O(\epsilon)$. Equation 2.24 is fulfilled with probability of at least $1-\delta$, where δ is the probability from where the values of A were drawn, in Johnson-Lindenstrauss Lemma it is a Gaussian distribution. The JL lemmas is as follows:

For any $0 < \epsilon < 1/2$ and any integer $N > 4$, let $R = \frac{20\log(N)}{\epsilon^2}$. Then for any set X of N points in $\mathbb{R}^d$ $\exists\, f : \mathbb{R}^d \rightarrow \mathbb{R}^R, \quad \forall x_i, x_k \in X$

$$(1-\epsilon)\|x_i - x_k\|^2 \leq \|f(x_i) - f(x_k)\|^2 \leq (1+\epsilon)\|x_i - x_k\|^2 \text{ with } i \neq k \tag{2.24}$$

Equation 2.24 can be rewritten by taking the upper bound and reinterpreting $f(x)$ as a matrix vector multiplication.

$$\|A(x_i - x_k)\|_2 = (1+\epsilon)\|x_i - x_k\|_2, i \neq k \tag{2.25}$$

It can also be seen as a dot product

$$|\langle Ax_i, Ax_k\rangle - \langle x_i, x_k\rangle| \leq \epsilon\|x_i\|\|x_k\| \tag{2.26}$$

In a similar way the Euclidean distance between the two vectors is scaled by a factor of the previous dimension and the new dimension

$$\sqrt{d/R}\|(Ax_i - Ax_k)\|_2 \tag{2.27}$$

Equations 2.25 and 2.27 are equivalent:

$$\begin{aligned}\langle Ax_i, Ax_k\rangle &= (\|A(x_i + x_k)\|_2^2 - \|Ax_i\|_2^2 - \|Ax_k\|_2^2)/2 \\ &= (1 \pm \epsilon)\|x_i + x_k\|_2^2 - (1 \pm \epsilon)\|x_i\|_2^2 - (1 \pm \epsilon)\|x_k\|_2^2 \\ &= \langle x_i, x_k\rangle \pm O(\epsilon)\end{aligned} \tag{2.28}$$

In this case, the columns of matrix A, are drawn from a Gaussian distribution. Therefore, strictly speaking these matrices are not orthogonal and highly dense. However, considering that in a high dimensional space exists much larger number of almost orthogonal than non orthogonal directions. Thus A is close to orthogonality [9].
To calculate all pairwise distances for the input space we would need we need to compute XX^T which has a cost of $\mathcal{O}(N^2 d)$ [22]. This value is relevant when we compare to other types of Random Projections.

Different Types of Random Projections

What tells apart one type of Random Projection from another are the distribution from where the elements of the column vectors of A are drawn, the number of non-zero elements

and how the position of this elements inside $\vec{a}$ is chosen. The most common Random Projections are

1. **Dense Random Projections** produces the entries of A as iid drawn from $\mathcal{N}(0, \sigma^2)$. The variance can change, however it is recommended to chose $(a)_{i,j}$ in such a way that it follows a symmetric distribution [22]. This type of Random Projection reduces the computation of pairwise distances to $\mathcal{O}(NdR + N^2R)$.

2. **Sparse Random Projections** were introduced by Achlioptas [1] in 2003. The column vectors of random matrix consist of iid entries drawn as follows

$$(a)_{i,j} = \sqrt{s}\begin{cases} 1 & \text{with probability 1/(2s).} \\ 0 & \text{with probability 1 - 1/s.} \\ -1 & \text{with probability 1/(2s).} \end{cases} \tag{2.29}$$

   With, $s = 1$ or $s = 3$. When the latter is used a threefold speedup is obtained [22].

3. **Very Sparse Random Projections** applies equation 2.29 with $s = \sqrt{d}$ or $s = \dfrac{d}{\log d}$ which are significantly higher values. This adds more speedup to the algorithm with minimum loss of accuracy [22].

2.4. Locality Sensitive Hashing

Locality Sensitive Hashing (LSH) is a compilation of techniques that look for the *Approximate Nearest Neighbor* search or more specifically, the (R, c)-near neighbors of a given object(*query*). It was introduced by Indyk et al [19] in an attempt to remove the Curse of Dimensionality. It is based on a family of hash function $\mathcal{H}$ whose main objective is to hash similar instances into same codes with a higher probability than dissimilar items [35]. The widely used metrics to measure quality of results are *Precision* and *Recall*. Also other metrics like execution time and storage requirements are used for data base search. Its main disadvantages are the total probabilistic nature of the algorithm and its independence of input data. The later makes the algorithm to totally neglect the distribution of data [21]. In practice this algorithm is used for data base search, duplicate detection, clustering, image detection etc.

LSH is part of Approximate Nearest neighbor search, it requires a distance metric to calculate similarity between candidate pairs. For this reason many $\mathcal{H}$-families for different metrics have been develop, it all depends on the space where the data is. The formal definition of a family is a follows [35]

For two instances q and p a family of $\mathcal{H}$ is called (R, cR, P_1, P_2)-sensitive if any of the two instances:

$$\begin{aligned} &\text{if} \quad dist(p,q) \leq R, \quad \text{then} \quad Prob[h(p) = h(q)] \geq P_1, \\ &\text{if} \quad dist(p,q) \geq cR, \quad \text{then} \quad Prob[h(p) = h(q)] \leq P_2 \end{aligned} \tag{2.30}$$

Where, $c > 1$ and $P_1 > P_2$. One parameter to measure performance of an $\mathcal{H}$-family is ρ, it is defined as

$$\rho = \frac{\log(1/P_1)}{\log(1/P_2)} \tag{2.31}$$

The smaller ρ is the better performance the family shows. The theorem also states that for a given LSH family there exist (R, c)- nearest neighbor search problem which uses $\mathcal{O}(dN + N^{1+\rho})$ space, with $\mathcal{O}(N^{\rho})$ distance computations and $\mathcal{O}(N^{\rho} \log_{1/P_2}(N))$ evaluations of hash functions [35].

LSH uses tables compounded of *Buckets*. These bucket are hash codes where x_i are hashed, in other words, each bucket is an $h(x_i)$ result. In this case, LSH tries to maximize collisions between similar instances, considering all instances in same bucket as candidate pairs. At the beginning of the algorithm there are as many buckets as instances, however, not all of them are output, only the non-empty buckets are considered.

Taking into account that $P_1 > P_2$ it is necessary to close the gap between those two probabilities by concatenating many hash functions i.e. $J(x_i) = \{h_1(x_i), h_2(x_i)..., h_g(x_i)\}$. All g-number of functions are drawn uniformly at random from $\mathcal{H}$.

There exist different LSH-families that are chosen depending on the distance metrics that it uses. In this case we focus on l_p distances. Therefore, only this family of LSH function is presented in the next section. To a more detail description of other LSH families you can refer to Wang et al [35] who made an extensive review of LSH algorithm.

2.4.1. l_p-LSH Family

This family of hash functions are used when an l_p metric is chosen. In this case we are finding the similarity with $\|x_i - x_j\|_p$ with p taking real values between 0 (exclusive) and 2 (inclusive). The most known l_p distances are the Manhattan distances ($p = 1$) and the Euclidean distance (p=2). For the former $\rho = \frac{1}{c} + \mathcal{O}(\frac{R}{r})$, where r is the radius or distance threshold and $c = \|x_i - x_j\|_1$. For the later, $\rho < \frac{1}{c}$ for an r carefully chosen [35].

The hash function for this type of LSH-family is shown in equation 2.32. Where, $b \in [0, r]$ and $\vec{w}$ is the projection plane drawn from a Gaussian distribution. In some applications

$b = 0$ to save storage.

$$h_{\vec{w},b} = \frac{w^T x + b}{r} \tag{2.32}$$

2.4.2. Distance Metrics in High Dimensional Spaces

This section is based entirely on the research done by Aggarwal et al [2], where he exposed some interesting properties of l_p-distances when they are used in high dimensional spaces. It has been found that on spaces with high dimensions for a given query (q) the distance between the nearest neighbor to the farthest is almost the same. This make the similarity algorithms to look almost every instance in the data set. It also makes the selection of a distance metric more challenging than in lower dimensional spaces.

Aggarwal et. al. shows how $p = 1$ gives better results than $p \geq 2$. They define *Fraction distance metrics* for $p \in (0, 1)$ (see Equation 2.33) as the best way to find distance in a high dimensional spaces. However, using Equation 2.33 represents adding more computational cost to distance calculation. Therefore, a trade-off needs to be found between cost and accuracy.

$$dist_d^p = \sum_{i=1}^{d} [(x_i - y_i)^p]^{1/p} \tag{2.33}$$

Where, x_i and y_i are the values of instance x and y in the i-th dimension. And d is the number of dimensions.

Part II.

Implementation and Numerical Experiments

3. Pipeline

In his section all details related to the pipeline implementation are presented. The whole pipeline consist of 4 main stages: LSH with Random Projections, dimensionality reduction of clusters and Sparse Grid density Estimation. See flowchart in figure 3.1; For the sake of simplicity of the flowchart some intermediate calculations, as well as, quality metric computation for each step are omitted. These metrics are shown in the subsection corresponding to that phase.

The main goal of this pipeline is to implement a Locality Sensitive Hashing (LSH) with Random Projections (RP), together with an agglomeration algorithm. Then, apply dimensionality reduction to see how the performance of Sparse Grid Density Estimation($SGDE$) is improved. The improvement is measure in terms of Grid Points Density(GPD), Elapse time(t) and *Log-likelihood* ($\mathcal{L}$) of test data. These three are used to compare the results of our approach with the conventional application of SGDE (density estimation of the whole data set at once).

The preprocessing step is the conventional instance-wise normalization of data done before before applying any machine learning algorithm. All other phases and their pseudocode are shown in their corresponding section.

The main objective of this thesis is to test the application of SGDE to clusters[1] individually. These clusters are selected using Random Projections and LSH and then dimensionally reduced. For that reason the left branch of **if condition** (condition evaluated to `True`) in figure 3.1 is the one explained in detail. Furthermore, there are plenty of literature where the conventional approach (`False` branch) is analyzed or implemented: Bungartz et al [10], Peherstorfer [25], Peherstorfer et al [26], Garcke et al [14], [15]. However, both approaches are executed in this thesis for comparison purposes (See Numerical Experiments chapter).

3.1. Random Projection

This step is applied to the whole preprocessed set ($\hat{X}$) where a predefined set of parameters like reduced dimension (R), epsilon (ϵ), `seed` and type of Random Projection (`type_p`) are input to the algorithm. Out of the four parameters, `seed` and `type_p` are found using

[1] clusters are also referred in this chapter as groups

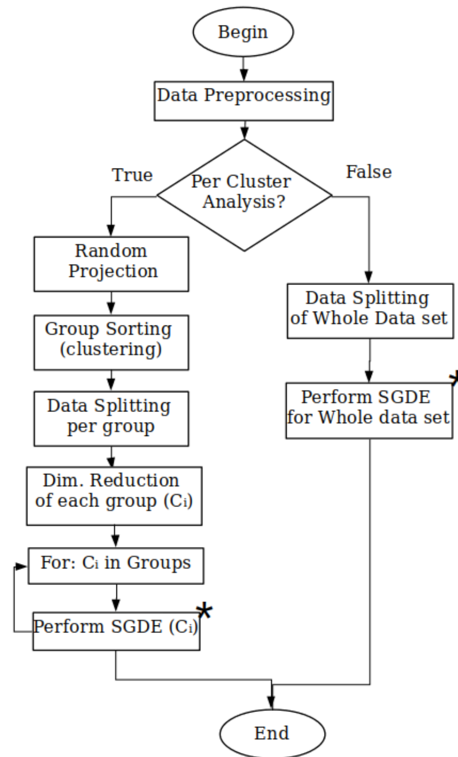


Figure 3.1.: Pipeline flowchart: on the **false** branch we see the conventional application of SGDE i.e. Density Estimation applied to the whole data set. The **True** branch is the approach implemented in this project. Where, the Random projection and group sorting belong to LSH process. Data splitting stage makes reference to the division of input set into train and test sets. Dimensionality reduction is applied to each clusters c_i independently. Finally, SGDE is applied to each cluster. One important remark is the $\star$, it means that the actual computational task is the same, what changes is the input data and some grid modifications that we show in later sections.

an iterative approach, the remaining two parameters are given by the user. Go to Dimensionality Reduction in Theory chapter for formal the definition of each of those parameters.

For the actual projection the library **random_projection** by **scikit learn** [31] is used. There are two functions in this library that were used: `GaussianRandomProjection` and `SparseRandomProjection`, in our case, these are referenced as `type_p`.

Each column vector of a Random Projection can be interpreted as a *Hash function* ($h(\cdot)$) [24] and it is used in the **Random hyperplanes** technique [21]. The entries of $h(\cdot)$ using `scikit-learn` library are drawn either from 3.1 for a Gaussian Random Projection or from 3.2 when it is an Sparse Random Projection.

The pseudocode for this step is shown in 1. The function `johnsonLindenstrauss(N,`ϵ`)` uses the `Johnson-Lindenstrauss Lemma` [20] to recommend a minimum dimension for the manifold embedding considering N instances and the acceptable distortion (ϵ) of the pair-wise distances when projected.

$$\{h_i\}_{i=1}^{N} = \mathcal{N}(0, \frac{1}{R}) \tag{3.1}$$

$$\{h_i\}_{i=1}^{N} = \begin{cases} +\sqrt{\frac{s}{R}} & \text{with probability } 1/(2s). \\ 0 & \text{with probability } 1 - 1/s. \\ -\sqrt{\frac{s}{R}} & \text{with probability } 1/(2s). \end{cases} \tag{3.2}$$

With, $s = \sqrt{d}$ is the number of non-zero entries in $h(\cdot)$ as recommended by Ping [22].

Algorithm 1 Random Projection

1: This function takes $\hat{X} \in \mathbb{R}^{N \times d} \rightarrow \hat{X}_{RP} \in \mathbb{R}^{N \times R}$
2: **procedure** PERFORMRANDOMPROJECTION($\hat{X}, \epsilon, R, type_p, use_R = True$)
3: $min_dim = johnsonLindenstrauss(N, \epsilon)$ → N is the number of instances
4: **if** $use_R == True$ **then** → Use the recommended minimum dim or the input one
5: $R = min_dim$
6: **end if**
7: **if** $type_p ==$ `'Sparse'` **then**
8: $\hat{X}_{RP} = SparseRandomProjection(R, random_state = seed)$
9: **else**
10: $\hat{X}_{RP} = GaussianRandomProjection(R, random_state = seed)$
11: **end if**
12: **end procedure**

3.1.1. Finding the Best Possible Projection

Due to the random nature of the elements contained in the projecting matrix a Random Projection can be highly unstable and could give very different results [6]. Therefore, it is necessary to find the best possible projection based on the `seed` and `type_p`. Consequently, for every data set one iterative algorithm that varies `seed` and `type_p` was implemented using 1. It does not ensures a global optimal projection but only a local one in the range of the given `seeds`.

In figure 3.2 we see an schematic example of two slightly different projections that lead to different results when applied to same data. An important remark is the behavior of the cluster radius h (see Figure 3.2) when projected: The better the projection, the smaller h tends to be. It is a good property that helps the proceeding steps. However, it is not always the case, a good projection could lead to higher values of h compare to a non-optimal projection. An indirect way to evaluate the aforementioned behavior of h is computing Recall and Precision after LSH application (see Section 3.2.3).

The metrics to evaluate the quality of a random projection are measured after clustering, these are: *recall per cluster*, *total recall*, *precision per cluster* , *execution time* and *amount of used memory* of the final clusters. All of these metrics are going to be introduced in following sections.

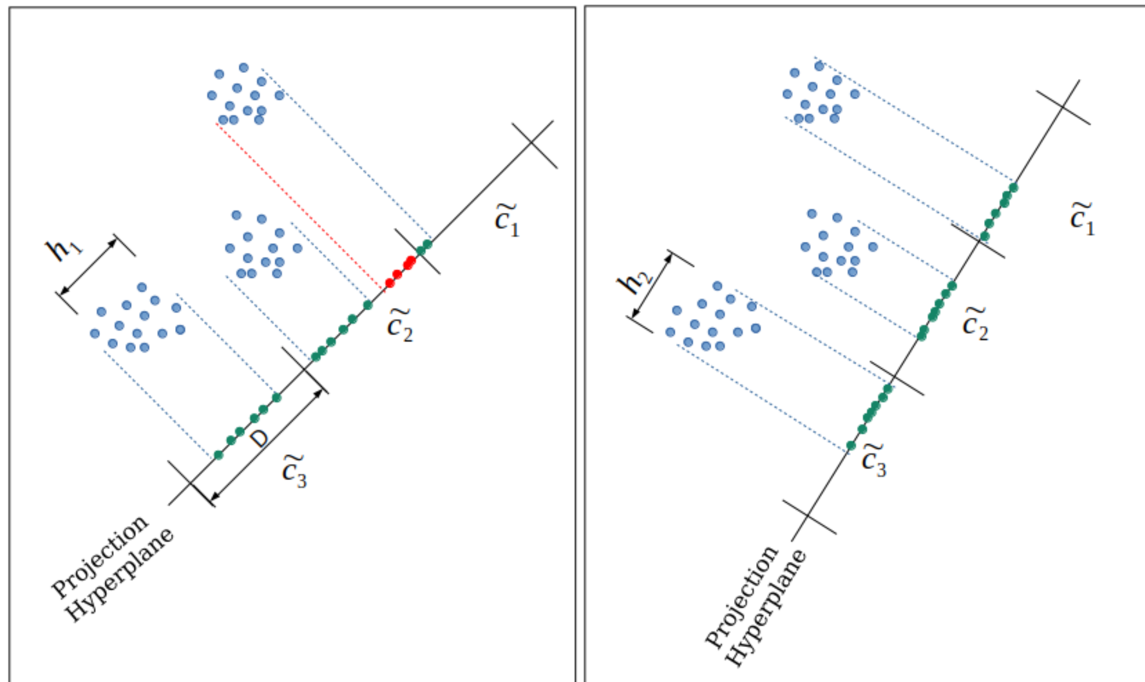
3.2. Group Sorting

Once $\hat{X}_{RP}$ is obtained, next step is to cluster all similar instances into groups[2]. To perform this clustering it is necessary to learn different parameters like distance threshold (D), number of bands*(B)* and Number of rows per band*(M)*. The distance threshold is used to ensure that all instances in one group are close enough to each other. The parameters B and M are used to hash the instances to their corresponding buckets, hence affecting LSH accuracy.

In the following subsections we show how the *sketch* [3] are build from the projected matrix, and how *D, B* and M are learned. Then, the LSH implementation with a second distance filter that uses the idea of *single-link clustering* is presented. Third, Quality metrics are introduced to quantify the output of the algorithm. And finally, the pseudocode for the whole group sorting part is exposed.

[2] clustering and group sorting make reference to same procedure

[3] An sketch is a discrete representation of the projected matrix that contains +1 and -1. It has the same dimensions as $\hat{X}_{RP}$



(a) Random Projection 1 with false positives in $\tilde{c}_2$ (red dots in projection hyperplane).

(b) Random Projection 2 with no false positives nor false negatives

Figure 3.2.: Shows the effect of randomness by comparing two Random projections for clusters $\tilde{c}_1$, $\tilde{c}_2$ and $\tilde{c}_3$. With distance D fixed for all clusters and $h_1 > h_2$. Dash lines indicate the projection of outlier samples that belongs to same cluster. Red dash line as well as red dots indicate a projection that causes *false positives* in $\tilde{c}_2$ and *false negatives* in $\tilde{c}_1$, hence, decreasing Recall and Precision of two neighboring clusters. On the other hand, green dots and blue dash lines denote a good preservation of cluster structure when projected. h_i represents the distance of the farthest points in same cluster in the projection plane.

3.2.1. Distance Metrics, Sketch Construction and Parameter Learning

Distance Metrics

Before starting sketch construction is necessary to define three distance metrics that are used in different parts of the implementation.

1. Jaccard Similarity(s), also called Jaccard Coefficient is a similarity measure between two sets, $\mathcal{A}$ and $\mathcal{B}$ defines in equation 3.3. The closer s is to one, the more similar $\mathcal{A}$ and $\mathcal{B}$ are. Hence, the distance between those two sets is computed as $1 - sim(\mathcal{A}, \mathcal{B})$ [21].

$$s = sim(\mathcal{A}, \mathcal{B}) = \frac{|\mathcal{A} \cap \mathcal{B}|}{|\mathcal{A} \cup \mathcal{B}|} \tag{3.3}$$

2. Cosine distances are used in Eucledian Spaces where each point is seen as a vector. This parameter measures the angle between $\vec{a}$ and $\vec{b}$. Therefore, the closer ϕ is to zero the more similar two instances are [21].

$$\begin{aligned} \cos_d(\vec{a}, \vec{b}) &= 1 - \frac{\phi}{180°} \\ \phi &= \cos^{-1}\left(\frac{a \cdot b}{\|a\|_2 \|b\|_2}\right) \end{aligned} \tag{3.4}$$

3. Out of all l_p-norms the most common ones are the Manhattan distance ($p = 1$) and the Euclidean distance ($p = 2$). In this document, when using an l_p-norm, we use the Manhattan distance. When $p = 1$ a good behavior in high dimensional spaces is obtained in contrast to norms with $p > 1$ [3]. It also represents a lower computational cost compared to *fractional distances* $(0 < p < 1)$.

$$\|\vec{a} - \vec{b}\|_p = (\sum_{i=1}^{d} \|a_i - b_i\|_p)^{(1/p)} \tag{3.5}$$

Notice the summation in equation 3.5 goes from $i = 1$ to d. It means that we use l_p-norms only in the input space. On the other hand equations 3.4 and 3.3 are going to be used only in the projected space, i.e. only applied in *sketch* or columns vectors of $\hat{X}_{RP}$.

Sketch Construction

Having $\hat{X}_{RP}$ with entries $(\hat{x})_{i,j} \in \mathbb{R}$ the construction of the *sketch* ($\hat{S}$) is done by replacing them with discrete values from $\{-1, +1\}$ [21], as follows

$$(\hat{S})_{i,j} = \begin{cases} +1 & \text{when } (\hat{x})_{i,j} \geq 0. \\ -1 & \text{when } (\hat{x})_{i,j} < 0. \end{cases} \tag{3.6}$$

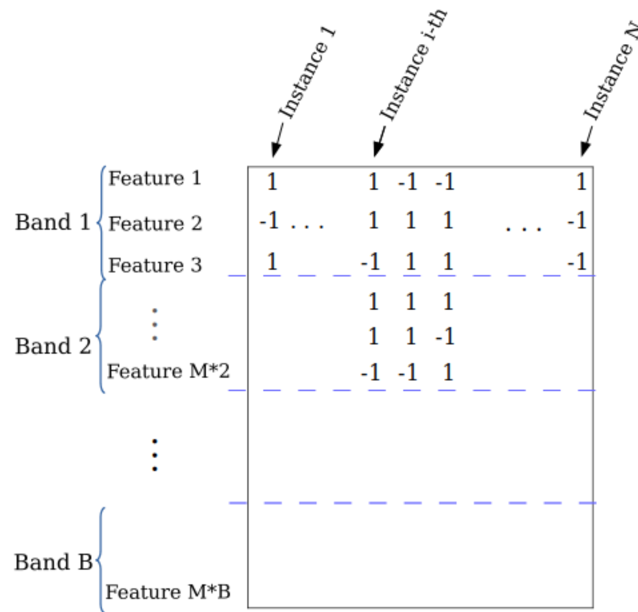


Figure 3.3.: Sketch banding process is the division in B-bands that are composed by one or more M-row vectors of the sketch. In this example $M = 3$, features are row vectors and instances are column vectors.

Parameter Learning and Sketch Banding

For this procedure we follow the step shown in Leskovec [21]; With *sketch*'s dimensions $N \times R$, the number of rows per band times the number of bands must be equal to the dimension of the final embedding, i.e. $R = M \times B$.

First, let's remember the structure of the *sketch*, where the column vectors are projected instances to R-space. In other words, each entry of that matrix is the result of a hash function applied using a Random projection to input instances i.e. $h(\boldsymbol{x})$, followed by an element-wise application of 3.6. See figure 3.3 that illustrate the banding process.

To learn parameter M and B we use the result from the analysis made by Leskovec et. al. [21] in pages 107 and 108. We take the equation $1 - (1 - s^M)^B$ that gives the probability that two instances hash to same bucket in at least one band. Where, s is the minimum acceptable similarity between two instances. In an example given by same authors, if we choose $M = 5$, $B = 20$ and similarity $s = 0.8$, yields that only one in 3000 pairs are false negatives.

To learn D we did a qualitative analysis to select what is the best threshold between distances of samples in same cluster and distances in different clusters.

3.2.2. LSH Algorithm: Putting All Pieces Together

Starting from the Random Projection (or Random hyperplanes in this case) all through the banding of sketches, we were using an LSH algorithm. In this subsection, we are showing the remaining part of the clustering with LSH and finally an schematic of the process and the pseudocode.
An important difference from the conventional LSH to the one used here is the lack of a *query object* ($\vec{q}$). We do not probe for similar object to $\vec{q}$, but for all similar objects that hash to same bucket. An extensive review of LSH algorithms and its variants is given by Wang et al. [35]. The following procedure is taken form Leskovec et at. [21] in section 3.4.2 of his book, except step 4 and a slight modification in step 1.

Procedure

For a better understanding of the procedure see algorithm 2 and the schematic example for the first band computation and $M = 3$ in figure 3.4.

1. Using the banded sketch, we start hashing the entries for each instance in one band: computing the dot product with a vector containing the powers of two. The result of dot product is an integer that identifies an specific **bucket**. See equation 3.7. This computation is performed for all instances.

$$\begin{aligned} Bucket &= \sum_{i=1}^{M} \hat{S}_i \cdot y_i \\ \hat{S} &= (\hat{S})_i \text{ for } i = b \cdot M + 1, b \cdot M + 2, ..., (b+1) \cdot M \\ y &= (2^i) \text{ for } i = 0, 1, ..., M \end{aligned} \tag{3.7}$$

   Where, b denotes band index and goes from $b = 0, ..., B-1$ and $(\hat{S})_i$ is the i-th vector of sketch matrix. Notice that $\hat{S}$ is different to s which is the Jaccard similarity 3.3.

2. After all instances in one band are hashed we compare Buckets. If two instances hash to same bucket a **first distance filter** is applied using equation 3.5 with same instances in input space. If distance is smaller than a threshold then effectively put those two samples in same bucket, otherwise, discard that computation. Furthermore, after a band is entirely probed all empty buckets are discarded.

3. Steps (1) and (2) are perform for all bands. If two instances hash to same bucket in at least one band they are considered part of the same cluster. If this happens those two instances are not computed again for the remaining bands.
   After all bands have been processed, all remaining buckets are considered clusters

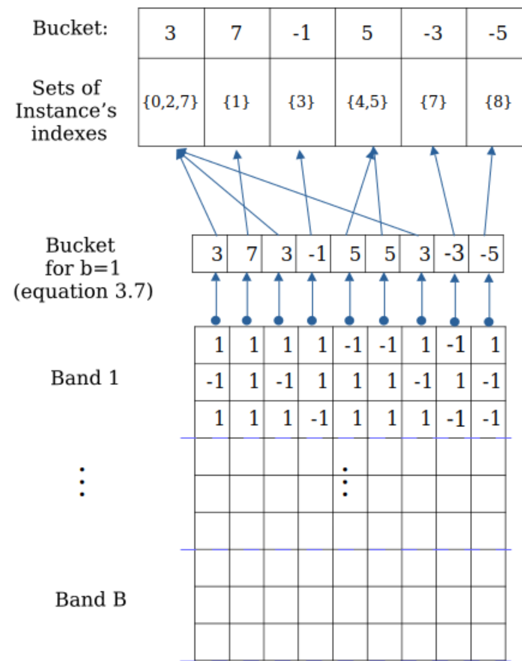
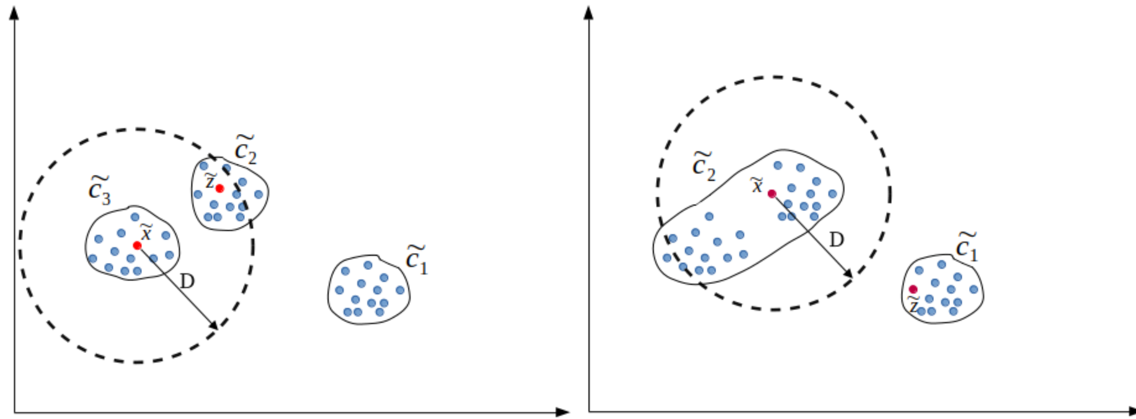


Figure 3.4.: LSH schematics for one band and $M = 3$. Let's consider the first column vector of band one ($b = 1$): $x^T = [1, -1, 1]$ as an example. Then, we apply the dot product with the powers of two ($1 * 2^0 + (-1) * 2^1 + 1 * 2^2 = 3$). We perform the same computation for all column vectors in $b = 1$. Finally, we take instances that hashed to bucket number 3. This corresponds to step (1) in the procedure above.



(a) iteration $i-1$. $\tilde{c}_3$ and $\tilde{c}_2$ are are effectively agglomerated. Then, clusters are renumbered for iteration i.

(b) iteration i. Test condition is violated this time. Therefore, no change in $\tilde{c}_1$ and $\tilde{c}_2$. For this iteration new random samples are selected.

Figure 3.5.: 2-Dimensional representation of two iterations for Agglomeration algorithm (second filter). Where $\|\cdot\|_p$ tests if $\tilde{\boldsymbol{z}}$ falls inside an hypersphere centered at $\tilde{\boldsymbol{x}}$ of radius D. Red dots are the two random samples from each clusters under comparison.

or groups denoted as $\tilde{c}_i \in \tilde{C}$ for $i = 1, ..., \tilde{k}$. Where, $\tilde{k}$ is the number of found clusters after step (3).
The number of available buckets before starting at step (1) has to be (2^M) for each band and $((2^M) \times B)$ for the whole algorithm.

4. Finally, a **second distance filter** between groups $\tilde{c}_i$ is performed using once more equation 3.5 , in this case *single-link clustering* is the method of choice as agglomeration algorithm(see equation 2.10 from Hierarchical clustering). To agglomerate the clusters two random vectors $\tilde{\boldsymbol{x}} \in \tilde{c}_i$ and $\tilde{\boldsymbol{z}} \in \tilde{c}_j$ where $i \neq j$ are selected and tested for $\|\tilde{\boldsymbol{x}} - \tilde{\boldsymbol{z}}\|_p \leq D$. If the condition is fulfilled the two clusters are agglomerated. See schematic example in figure 3.5.

   One important remark is the possibility of *"propagation"* of the agglomeration [5], wrongfully leaving only one cluster after the computation is finished. Therefore, distance threshold D needs to be carefully chosen .

   Strictly speaking, this step is not part of LSH algorithm but of nearest neighbors search ($K-NN$). We added this small modification that improves LSH results with an extra computational cost of only $\mathcal{O}(\tilde{k})$ comparisons.

Algorithm 2 Complete LSH implementation

This function hashes instances from $\hat{X}_{RP} \in \mathbb{R}^{N \times R}$ into clusters/groups.

2: **procedure** PERFORMLSH($\hat{X}, \hat{X}_{RP}, B, M, D$)
  $(\hat{S})_{i,j} = \{+1, -1\}$ → Sketch computation (Same size as $\hat{X}_{RP}$).
4: $\tilde{C}[0:(2^M) \times B] = \{\emptyset\}$ → Array of sets. Total number of buckets a.k.a clusters
  $y[0:M] = (2^i)_i$, for $i = 0, ..., M$
6: **for** (b=0:B-1) **do**
   $Bucket's = \hat{S}[b \times M : (b+1) \times M] \cdot y[0:M]$ → Hash instances to buckets
8:   **for** (i=0:length($Bucket's$)) **do**
    if (Exists $Bucket[j] == Bucket[i], i \neq j$) OR ($i \in \tilde{C}$) **then**
10:     Break → Do not compute already hashed instances
    else
12:     **if** $\|\hat{X}[i] - \hat{X}[j]\|_p < D$ **then**) → Instances i and j in same Bucket
      $\tilde{C}[i] = \{i, j\}$ → i and j are considered to be in same cluster
14:     **end if**
    end if
16:   **end for**
  end for
18: delete $\tilde{C}[\tilde{k}]$ $\forall$ $\tilde{k}$ such that $\tilde{C}[\tilde{k}] = \{\emptyset\}$ → Delete empty clusters/buckets
  k ← length($\tilde{C}$)
20: **for** (i=k-1:0) **do** → Second distance filter between found clusters is applied
   for (j=i-1:0) **do**
22:    $\tilde{x} \leftarrow \hat{X}[v]$; v randomly selected from $\tilde{C}[i]$ indexes
    $\tilde{z} \leftarrow \hat{X}[w]$; w randomly selected from $\tilde{C}[j]$ indexes
24:    **if** $\|\tilde{x} - \tilde{z}\|_p < D$ **then**
     $\tilde{C}[j] = \tilde{C}[i] \cup \tilde{C}[j]$
26:     delete $\tilde{C}[i]$
     Break
28:    **end if**
   end for
30: **end for**
  return $\tilde{C}$
32: **end procedure**

3.2.3. Validation Metrics for LSH

Up to now we do not have any indicator of the pipeline performance regarding correctness of the found groups. In this section we introduce four different metrics that let us quantify that performance: Recall, Precision, Average Recall and Number of missed instances.

Let's remember that the final output of algorithm 2 is a list of sets(c_i), in which each c_i contains indexes that presumably belongs to one specific cluster. With this list and the ground truth (I_i), we can calculate the fraction of relevant indexes that were correctly selected per cluster i.e Recall.

$$Recall_i = \frac{|\{c_i\} \cap \{I_i\}|}{|\{I_i\}|} \tag{3.8}$$

And the fraction of selected indexes that are actually relevant per cluster i.e. Precision.

$$Precision_i = \frac{|\{c_i\} \cap \{I_i\}|}{|\{c_i\}|} \tag{3.9}$$

In case there is no ground truth to calculate this fraction one can use the output of any exact *k-NN* algorithm [24].

There are some cases in which 3.9 and 3.8 show an acceptable result but the algorithm find more groups than actually exist. Therefore, we use the average Recall to punish the algorithm performance by dividing the sum of Recalls by k (number of found groups) and setting Recall of false clusters to zero (see Equation 3.10).

$$avg_recall = \frac{1}{k}\sum_{i=1}^{k} Recall_i \tag{3.10}$$

Finally, there can also be cases in which instances are not hashed to any group (this is an undesired consequence of the first distance filter). Therefore, we report this metric as a integer denoted as $\boldsymbol{m}$. To continue with the pipeline we put the m instances into the more similar cluster by calculating l_p-norm for each of them (same idea as second distance filter).

3.3. Dimensionality Reduction of Clusters

Sparse Grid Density Estimation show better performance than other algorithms when it is applied to highly-dimensional data sets. However, It does not tackle entirely the *Curse of Dimensionality*. Therefore, in our implementation is necessary to perform some previous steps to make feasible SGDE computations. In this section, we show how each group,

found in LSH implementation, are reduced to an space of significantly lower dimension.

The Random projection shown in section 3.1 and LSH in general was done in order to find the clusters i.e. put instances in groups that are closer to each other compared to instances in other groups. However, these clusters are still in the input space which is of high dimension. We have to perform an extra step of feature extraction to reduce dimensionality.

The projection of clusters to a lower-dimensional space is done individually using either *PCA* only or an hybrid approach of *Random Projection* plus *PCA*. In the former, PCA takes iteratively the clusters c_i for $i = 0...k$ from the input dimension d to the reduced dimension q. In the latter, we take the clusters iteratively and reduced them to the minimum recommended dimension according to *Johnson-Lindenstrauss Lemma* using an *Sparse Random Projection*, then PCA to the final dimension q.

The used library for the Random Projection part is `sklearn.random_projection` and the functions `SparseRandomProjection` and `johnson_lindenstrauss_min_dim`. And The `PCA` funtion from `sklearn.decomposition` library. Both libraries by **scikit learn** [31]. The source code for this procedure is shown in the code snippet 3.1.

3.4. SG++ Library

In this section, first we give a brief review of what SG++ is, focusing on the most relevant classes and methods for our implementation. Second, the general settings needed by the library are introduced. Third, we focus on how grid is customized in our implementation using the already available methods in SG++. Finally, evaluation metrics are defined. Some code snippets for the most relevant parts are shown through the whole section.

SG++ is a C++ open-source library that is being expanded and maintained at the Technical University of Munich and the University of Stuttgart. It was develop by Dirk Pflüger in 2010 [33] [27] . This library contains different modules for function interpolation, Data mining and Machine learning, Partial Differential Equations, Uncertainty Quantification, Function optimization etc. We mostly use the base features for grid creation and customization, data structures, uncertainty quantification tools and the conjugate gradient solver.

The most relevant modules for us are:

1. **sgpp::base** namespace includes `sgpp::base::Grid` class that creates the grid according to user input. It has 33 different types of grids that can be created e.g. Linear grid, Linear Clenshaw Curtis grid, Square Root grid etc. From all theses the one we use is the linear grid without boundary points (see section 3.4.2 for more details).

```
from sklearn.random_projection import johnson_lindenstrauss_min_dim
from sklearn.random_projection import SparseRandomProjection
from sklearn.decomposition import PCA
import numpy as np

def perform_PCA(cluster, q, seed_PCA):
 #pca is an object of type decomposition.PCA
 pca = PCA(n_components=q,random_state=seed_PCA )
 #Here the actual dim reduction is perform:
 # From cluster.shape[1] to q
 projected_data=pca.fit_transform(cluster)
 return projected_data

def perform_RP(cluster, seed_RP):
 #"min_d" denotes the minimum recommended dimension
 #using Johnson-Lindenstrauss Lemma
 min_d = johnson_lindenstrauss_min_dim(cluster.shape[0],eps=0.1)
 #RP is an object of type RandomProjection
 RP=SparseRandomProjection(n_components=min,random_state=seed_RP)
 #Here the actual projection is perform
 projected_data = RP.fit_transform(cluster)
 return projected_data

for i in range(k):
   #cluster_idxs is the set of indexes for one cluster.
   #Data is the whole input data set.
   cluster_idxs=C[i]
   cluster_samples=data[np.array(list(cluster_idxs)).astype(int),:]

  if args.mode=='hybrid':
    #random projection to minimum recommended dim by JL lemma
    cluster_RP=perform_RP(cluster_samples,seed=seed_RP)

    #Dim. Reduction from minimum recommended dim by JL lemma to q
    cluster_PCA=perform_PCA(cluster_RP,q,seed_PCA)
  else:
    #Dim. Reduction from d to q
    cluster_PCA=perform_PCA(cluster_samples,q,seed_PCA)
```

Source Code 3.1.: This script takes c_i from d-dimension $\rightarrow$ q-dimension, where $q << d$. C is a list of sets, where, each set is a cluster that contains the instance's indexes that belong to that cluster. For simplicity, all code related to util functions (save logs, plotting etc) is omitted.

It also provides data structures like Data Vector, Data Matrix, Basis functions(e.g. piecewise linear/polynomial, B-splines) and operations e.g. adaptive refinement and multiple point evaluations.

2. **sgpp::datadriven** module provides functionalities for machine learning and data mining. The most relevant for us is the Uncertainty Quantification (uq). This includes a density estimation interface implemented in the classes `sgpp::datadriven::KDE` and `sgpp::datadriven::SparseGridDensityEstimator` just to mention some. The latter is actually the class we used for density estimation.

3. **sgpp::solver** module that implements different methods to solve linear systems of equations, PDE's, etc. This module is important for us due to the use of Conjugate gradient (CG) method that solves the system of equation necessary to estimate density. Solving the linear system represents the highest computational cost for the whole pipeline.

3.4.1. General Settings

We use the Python interface provided by SG++ in its Github repository [32] named `pysgpp`. Out of the different methods to perform density estimation we selected the instanciation of an SGDE distribution using config files (`SGDEdist.byLearnerSGDEConfig`) which takes as arguments:

1. samples: data set that contains instances row-wise

2. grid: it is an optional argument of type `sgpp::base::Grid`. It is used to estimate density based on a predefined grid instead of the one built internally based on the `config` argument. This is one of the main functionalities we are going to exploit in our implementation.

3. bounds: optional argument used in case the user wants to modify the bounds of the hypercube where the grid points are located.

4. unitIntegrand: optional argument used to force the results to have a unit integrant, which is not granted when using `identity` matrix as regularization type instead of laplace.

5. config: dictionary that should contain all the information regarding: grid (type, level and degree of basis functions), refinement configuration, solver settings (e.g. threshold for residual when using CG method), Cross validation settings etc.

In this method we made an small modification to use a customized grid and still use a config file for all remaining arguments. This modification consist of an `If` statement that checks the use of a grid with bounding boxes. In code snippet 3.2 you can see this modification. The rest of the code in this method stays the same as the one provided by SG++.

Also in the following section we show what are the modification done to the grid.

The main drawback for us regarding the conventional implementation (source code 3.2) is that `initialize(unit_samples_vec)` overwrites our grid that was previously customize with **bounding boxes**. Therefore, we create a data vector that contains all the weights computed by CG method and then we immediately start training. That solves the situation.

3.4.2. Grid Configuration

Considering that SG++ evaluates a function at every grid point(GP). Hence, the higher GP is, the more expensive the computation becomes. It represent a real constrain in terms of accuracy when more grid points are needed (adding more grid points usually leads to a better estimation).

We observed that in certain cases (e.g. when the data is clustered) some grid points are located in places where there is no actual data, it causes a waste of GP's. Therefore, by creating different localized grids for each cluster the number of unproductive GP's can be reduced. This opens the possibility to increase accuracy of SGDE with same number of grid points compared to conventional approach. Furthermore, in case there is no need for more accuracy, this methodology can help reducing the number of GP's (hence, computational cost). This also allows us to compute SGDE for each cluster in parallel.

For grid construction we used `sgpp::base::Grid` class to create a regular sparse linear grid with no boundary points. Then, we use the `sgpp::base::BoundingBox` class and the struct `BoundingBox1D` provided by SG++ to set the boundaries of each grid. We set boundaries for each bounding box as the minimum and maximum value of all samples in each cluster in every dimension. See code snippet 3.3 for grid customization. And in Figure 3.6 we show and example of Old faithful data set using the conventional approach(a) and using bounding boxes (b).

3.4.3. Validation Metrics for SGDE

To test the performance and accuracy after SGDE computation is finished we introduce Evaluation metrics for Density Estimation. This metrics aim to quantify any improvement or degradation of SGDE results using our methodology in comparison with conventional approach.

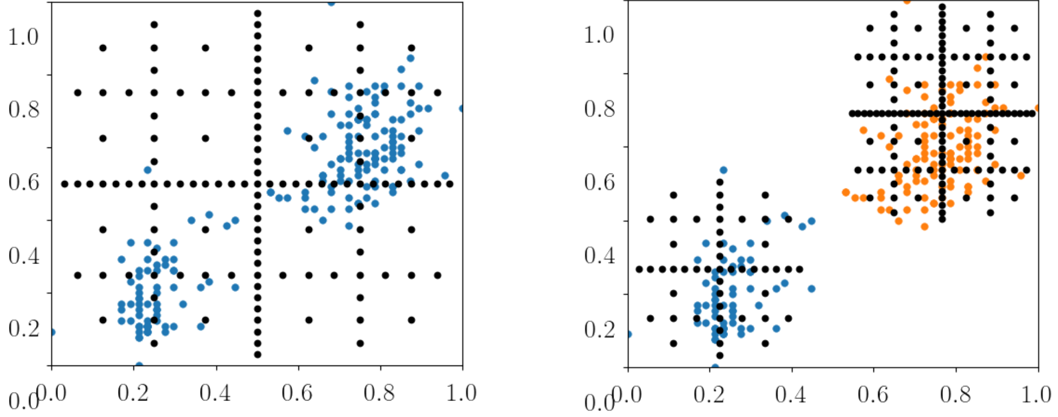
1. **Grid Point Density(GPD)**: Our implementation creates Grids contained in a volume smaller than the unit hypercube.Therefore, it concentrates more grid points where

```
learnerSGDEConfig =
    SparseGridDensityEstimatorConfiguration(filename_config)

learner = SparseGridDensityEstimator(learnerSGDEConfig)

#-----Added IF branch---------
#unit_samples_vec: data set
#lambda_reg: regularization parameter
#(see Theory chapter in SGDE section)
if bbox:
   # Creates a data vector of same number of elements as GP's.
   alpha=DataVector(np.ones(grid.getSize()) / 3.)

   # Start training using customized grid and alpha
   #learner.train() start CG method. It finishes when converged
   learner.train(grid,alpha,unit_samples_vec,lambda_reg)

   # Makes a copy the weights computed in the training step
   alpha_after = np.array(alpha.array())

else:
   # This is the conventional implementation for this method
   learner.initialize(unit_samples_vec)
   grid = learner.getGrid().clone()
```

Source Code 3.2.: Lines (2), (4) and the `else` branch is the conventional method provided by SG++:in(2) it creates the learner config file then it instantiate a learner of type **SGDE**(line 4). In line 19 the method **initialize()** set all parameters, create the grid and start the training. The `True` branch was added by us to avoid the creation of the grid internally but to use a previously customized one. That is why we do not call **initialize()** method but immediately the **train()** step.

```
import numpy as np
from pysgpp import Grid, BoundingBox,BoundingBox1D

def get_grid(dim,level,samples):
 # Creates a regular linear sparse grid in the unit domain
 # dim-dimensional sparse grid
  grid = Grid.createLinearGrid(dim)
  grid_gen=grid.getGenerator()
  grid_gen.regular(level)

 # looks for the minimum and maximum value of samples
 # in each dimension.
 # samples: Numpy array of samples from one cluster
  bb=[]
  for i in range(samples.shape[1]):
    bb.append([samples[:,i].min(),samples[:,i].max()])

 # Intantiate an object of type BoundingBox
 # and dimension = dim
    bb_ = BoundingBox(dim)

  for d_k in range(dim):
 # Declares and initialize struct of type BoundingBox1D
    dimbb = BoundingBox1D()

 # Sets boundaries in each dimension
    dimbb.leftBoundary = bb[d_k][0]
    dimbb.rightBoundary = bb[d_k][1]
    bb_.setBoundary(d_k, dimbb)

 # Compress grid to a Bounding box
  grid.getStorage().setBoundingBox(bb_)

  return grid
```

Source Code 3.3.: Code to create a grid and then compress it to a certain region of a unit domain marked by bounding box in every dimension



(a) Regular Sparse grid of level 5 with 129 GP (black dots) for the whole unit domain. We see that the top left quadrant has has almost no samples (blue dots) but still grid points.This leads to a not so efficient distribution of GP's

(b) Our implementations using bounding boxes with same data set as (a) but divided in two clusters. Blue cluster (blue dots) with a grid of level 4 with 49 GP and Orange cluster (orange dots) with 129 GP in a level 5 grid

Figure 3.6.: Comparative example between a regular sparse grid and two grids with bounding boxes for Old faithful data set.

the actual data is, giving a more efficient distribution of GP's.

It makes sense to define a parameter that describes how many grid points are contained in a volume unit. This parameter let us compare the two SGDE computations in terms of the number of GP's needed to obtain certain results in a given elapsed time. This metric is the Grid Point Density (GPD).

GPD is a ratio between the final number of grid points ($\#GP$) (including adaptive refinement) and the volume (or hyper-volume when $d > 3$) in which this grid points are distributed V(see Equation 3.11). When the computation is performed in the unit hypercube, this metric equals the number of grid points. On the other hand, for smaller domains, it increases; meaning a more dense computation.

$$
\begin{aligned}
GPD &= \frac{\#GP}{V} \\
V &= \prod_{i=1}^{q} |\max_j((c)_i) - \min_j((c)_i)|, \; with \; j = 1...N_T
\end{aligned}
\tag{3.11}
$$

Where, $\max_j((c)_i)$ and $\min_j((c)_i)$ means the maximum values and minimum values in dimension i for all N_T-instances in the input cluster c.

As an example, in Figure 3.6 (a) has a $GPD = 129$ and in (b) blue cluster $GPD = 49/(0.4468 * 0.5385) = 203.6$ and orange cluster $GPD = 447.8$. We see that blue cluster with 60% less grid points than (a), we get a GPD 58% higher. And in orange cluster with same level as (a) we get 3.4 times higher GPD than (a).

2. **Log-likelihood** of test data is calculated using the computed PDF evaluated at points $\boldsymbol{x} \in \mathcal{T}$ (Test set) [26]. The higher this metric is, the more accurate results are obtained from SGDE.

$$\mathcal{L} = \frac{1}{N_T} \sum_{x \in \mathcal{T}} log(p(\boldsymbol{x})) \tag{3.12}$$

Where, $p(\boldsymbol{x})$ is the estimated density evaluated at test sample $\boldsymbol{x}$.

3. **Elapse Time of Computation**(t) is the wall-clock time measured just before the creation of the grid until the Log-likelihood is computed.

4. Numerical Experiments

In this chapter we first present a summary of the input pipeline parameters, with a focus on the parameters that we vary when doing the numerical experiments. Second, the data sets used to test the pipeline is introduced, each of them with a brief description. Finally, all scenarios used to test the pipeline are shown.

4.1. Input Pipeline Parameters: Summary

In Table 4.1, you find a summary of all the input parameters for the whole pipeline. Most of them vary to build different scenarios as numerical experiments.

There exist more parameters for SGDE that do not play a relevant role in testing the implementation, therefore they are not mentioned in this table. These variables are either used with default values given by SG++ or are static in all scenarios. When the latter happens, we mention it explicitly in each scenario.

4.2. Data Sets

We use two data sets to test the pipeline. One is a synthetic data set and the other one a real-world data set. Both of them have clustered samples and have a high dimensionality ($d > 1000$).

Synthetic Data Set

The synthetic data is generated using a *scikit-learn* [30] library, called `datasets` with the function `make_blobs`. We generate a data set of $N = 900$ (instances) and $d = 10000$ (features). It contains 3 clusters, with 300 samples each. Furthermore, all samples in one cluster has an standard deviation of 12.

Real-World Data Set

For the real-world data set we use *expression cancer RNA-Seq Data Set* from *UCI-Machine Learning Repository* [4]. This data set contains 801 instances grouped in 5 clusters, every sample has 20531 features. In this case there is a class imbalance considering that each cluster represent one type of cancer: PRAD, LUAD, BRCA, KIRC, COAD containing 136, 141, 146, 78 and 300 samples respectively.

Item	Pipeline Parameter	Description
1	$\hat{X}$	Matrix — Pre-processed data set with dimensions $N \times d$. N-instances row-wise — All pipeline
2	`type_p`	String — Type of random projection: Sparse or Gaussian — Random Projection in LSH section
3	`epsilon`(ϵ)	Float — Acceptable pair-wise distortion for distances when projecting. If R is specified this parameter is ignored — Random Projection in LSH section
4	`R`	Integer — Dimension of final embedding. If ϵ is specified this parameter is ignored. — Random Projection in LSH section
5	`seed`	Integer — Input seed for Random Projection — Random Projection in LSH section
6	`B`	Integer — Number of bands for sketch banding — LSH
7	`M`	Integer — Number of rows per band for sketch banding — LSH
8	`D`	Float — Cluster radius for first and second filter — LSH
9	`mode`	String — Dimensionality reduction algorithm: PCA or Hybrid — Dim. Reduction
10	`q`	Integer — Final dimension after Dim. Reduction — Dim. Reduction
11	`parallel`	Boolean — True: Run SGDE in parallel (one cluster per process). Otherwise, SGDE for the entire data. — SGDE
12	`ref_Steps`	Integer — Number of refinement steps perform on the grid. — SGDE
12	`cluster extraction`	String — Methodology to obtain clusters in the reduced space — SGDE

Table 4.1.: Pipeline input parameters. The *Description* column has the format: Type of input — Description — Algorithm in pipeline that uses that parameter.

4.3. Numerical Experiments

There are two big parts in numerical experiments. One to test from the beginning up to LSH output and the second one test dimensionality reduction and SGDE implementation.

All numerical experiments were computed using a machine with Intel Xeon CPU E5-2697 v3 @ 2.60GHz, for notation simplicity we call it node 1. The number of cores used for

each scenario change, therefore we explicitly mention the number of cores used for each case. Furthermore, it is worth mentioning that in the whole pipeline the most expensive part to compute is the SGDE algorithm and that it is CPU-bound.

4.3.1. LSH Numerical Experiments

The first part test the implementation of LSH algorithm, including finding the best Random Projection. That means from section 3.1 to section 3.2 in implementation chapter. To quantify the result we use the metrics introduced in 3.2.3.
This part represents a negligible computational cost, hence it is computed sequentially using one core.

For this purpose we use parameters 2 to 8 in table 4.1, setting $D = 4000$ for Gene data set ($\hat{X}_{gen}$) and $D = 9000$ for synthetic data set ($\hat{X}_{syn}$). For both data sets:

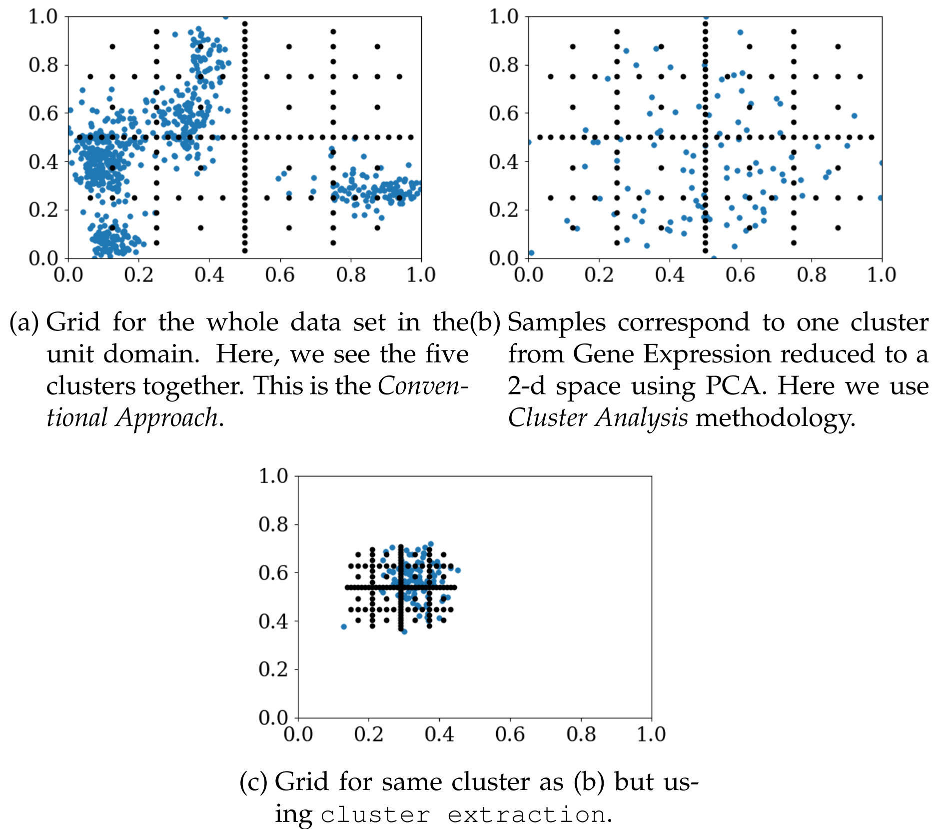
1. `type_p` is either Random or Sparse.
2. `R=100` therefore ϵ is an observed variable but not input.
3. `seed` ranges from 0 to 100.
4. `B=20` and `M=5`.

This results in 200 scenarios for each data set. The main goal for this experiment is to find the configuration for LSH parameters that optimizes the metrics shown in 3.2.3. After obtaining that configuration, we keep those parameters fixed and perform the second part of numerical experiments with that optimal settings.

4.3.2. Dimensionality Reduction and SGDE Numerical Experiments

In these experiments we test from section 3.3 to section 3.4.2 using metrics in 3.4.3 in Implementation chapter. In total there are 24 scenarios (12 experiments each data set).
For this, we use parameter 9 to 12 in table 4.1. These variables take values as follows:

1. `mode` is either PCA only or Hybrid (Random Projection then PCA).
2. `q=15` for the whole data set and each cluster.
3. `ref_Steps` is either 3 or 4 refinement steps. Each refinement with up to 100 grid points.
4. `parallel`, when `True` there is one process per cluster all computing in parallel. Otherwise, one process computes SGDE for the whole data set at once. Each process uses a defined number of cores.



(a) Grid for the whole data set in the unit domain. Here, we see the five clusters together. This is the *Conventional Approach*.

(b) Samples correspond to one cluster from Gene Expression reduced to a 2-d space using PCA. Here we use *Cluster Analysis* methodology.

(c) Grid for same cluster as (b) but using `cluster extraction`.

Figure 4.1.: Gene Expression Data set projected to a 2-dimensional space using PCA. Blue dots represent samples and black dots are grid points from a 4-level grid. The main difference between grids is the domain used for the computation.

5. `cluster extraction` in these cases we run 8 scenarios, all with `parallel=True` and `mode`, `q` and `ref_Steps` changing as mentioned above. `cluster extraction` refers to the way we extract cluster samples to compute SGDE. In the first 16 scenarios we vary parameters 9 to 12 as explained in the implementation chapter (see See plots (a) and (b) in Figure 4.1). In `cluster extraction` cases we reduce dimensionality of the whole data set to `q`, then we extract the samples that correspond to the indexes c_i found in LSH algorithm. See plot (c) in Figure 4.1.

In figure 4.1 we see a comparative example between different grids used for SGDE computations for these numerical experiments. These Figures show Gene Expression data set in a 2-dimensional space, which is actually not the space we use, however, we include them to guide the reader in the different types of computations that are performed.

We use a linear regular sparse grid with level three with linear basis functions. The regularization term is the Identity matrix with regularization parameter for both data sets as $\lambda = 10^{-5}$. However, as mentioned by Peherstorfer et. al [26] SGDE method is not very sentitive with respect to λ.
All other input parameters for SGDE are left as the default ones given by SG++ library [33].

Part III.

Results and Conclusions

5. Results and Analysis

In this chapter we present the results obtained by running all scenarios shown in 4. First we present results for LSH implementation and then results for SGDE. In both parts we divide them in Gene Expression data set and Synthetic data set subsections.

5.1. LSH Results for Gene Expression Data Set

After running all 200 scenarios explained in 4.3.1 for Gene Expression, the elapsed time for the whole LSH algorithm is between 27.4 to 42.9 seconds each scenario when using node 1 in one core. Elapsed time includes the Random Projection part which takes 1.6 to 2.3 seconds with no noticeable time difference when using `Sparse` or `Gaussian` projections.

In figure 5.1 we present the obtained average recall (equation 3.10) plotted against `seed`. In that plot we see how the random nature of the Random Projection affects the results with no clear pattern regarding average Recall. One can also concludes that neither `Sparse` nor `Gaussian` projections offer an obvious advantage when measuring the Recall of LSH implementation for this data set.
In these scenarios, our implementation reaches a maximum avg_recall of 95.15% and a minimum of 33.07% with some clusters getting $recall_i = 100\%$.

The average Precision versus the `seed` is presented in Figure 5.2. Once again, there is no clear advantage when using either of the two types of projections in terms of Precision. However, now we see that the majority of results are located in the upper part of the plot. This pattern allows to say that given a `seed` between 0 to 100, there is a high probability that the precision is higher than 80%, for this particular data set.
Our LSH implementation gets a maximum $Precision_i$ of 100% in some clusters and an average maximum Precision of 98.03% in all 200 scenarios.

The color bar in Figures 5.1 and 5.2 represents the number of missed samples (m). See at the end of section 3.2.3 for the definition of m. It also does not give us a clear winner between the two types of projections (`type_p`). In general, LSH implementation misses between 1.74% and 2.62% of samples.

Regarding `type_p` there is no clear choice when using the metrics introduced in section 3.2.3. However, in terms of used memory the `Sparse` random projection consumed

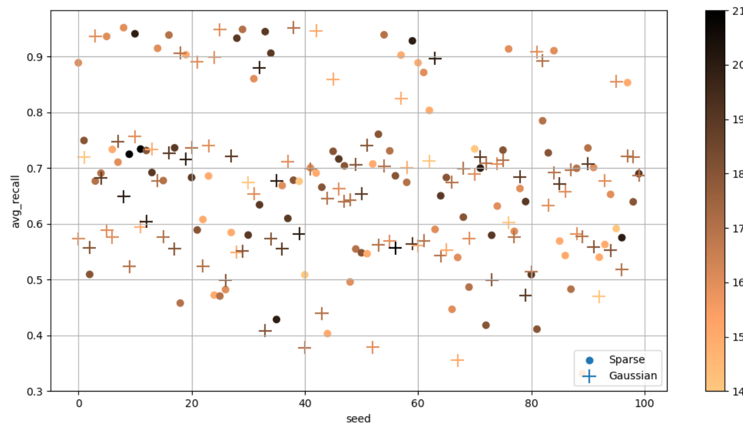


Figure 5.1.: In Y-axis is the avg_recall (equation 3.10) and in X-axis `seed` (input parameter). These are the LSH results using a Gaussian Random Projection(+) and a Sparse Random Projection (•). The color bar represents m (number of missed samples). We see that there is no clear advantage in using one type of projection over the other.

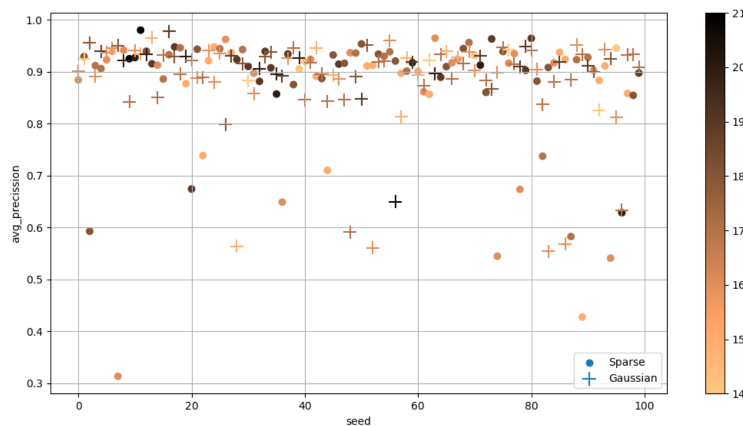


Figure 5.2.: $avg_precision$ (average over all clusters) vs `seed`. Both the Gaussian Random Projection(+) and a the Sparse Random Projection (•) tends to have a precision higher than 80% in most of the cases. The color bar represents m, which is not affected by `type_p`.

0.17±0.002 MB and the `Gaussian` projection used 0.32±0.001 MB. Therefore, for dimensionality reduction and SGDE experiments we set `type_p=Sparse`.

Furthermore, to select the `seed` we filter out all scenarios and chose the one with the the best trade-off between average Recall and Precision. With that we set `seed=8` for the remaining part of experiments in case of Gene data set. For this `seed` the average recall is 95.15% and the average precision 94.17% with $m = 16$.

5.2. LSH Results for Synthetic Data Set

In the case of synthetic data set, the total elapsed time is between 16.5 to 13.4 seconds. The Random Projection consumes 2.0 to 1.7 seconds of that elapsed time. Once more, there is no a relevant time difference between the two projection types when run in one core of node 1. In terms of memory, `Sparse` and `Gaussian` projection uses 0.12±0.01 MB and 0.36±0.001 MB, respectively.

For the Synthetic data set all scenarios result in $recall_i = 100\%$ and $precision_i = 100\%$. This is explained analyzing the structure of the data set in which every cluster is well defined around its centroid, with all samples within the cluster radius. Therefore, any random projection performed on these samples add some noise (ϵ), although cluster's structure is preserved.

Considering the results, to perform the second part of experiments for dimensionality reduction and SGDE all parameters are set to same values as Gene Expression Data Set (see section 5.1).

5.3. Dimensionality Reduction and SGDE Results for Gene Expression Data Set

The 12 experiments for Gene Expression data set for the Dimensionality Reduction part are run sequentially in one core of node 1. The elapsed time for `mode=PCA` is between 0.10 to 1.74 seconds and 2.93 to 12.38 seconds for `hybrid` mode. It shows a considerably longer elapsed times when using `mode=hybrid` (Random Projection then PCA).

For the SGDE computation, elapsed times are shown in Table 5.1. The difference in time between the different scenarios is due to the overhead of process creation and, mainly, adaptive refinement algorithm and Conjugate gradient method (CG) convergence. To be more specific let's make a brief refresher of how adaptive refinement works when using default values (Adaptive Refinement algorithm is embedded in SG++ library): After completing the first computation with zero refinement it has calculated a vector of sur-

Computation Type	`ref_Steps`	Minimum Elapsed Time [s]	Maximum Elapsed Time [s]	Average Elapse Time [s]
Whole Data set (`Parallel=False`)	3	4320	6537	5428
	4	9456	10568	10012
Per Clusters Analysis (`Parallel=True`)	3	6092	9934	8070
	4	14358	15788	15105
Cluster Extraction (`Parallel=True`)	3	4116	7011	5533
	4	9804	11885	10864

Table 5.1.: Elapsed times for all scenarios of Gene Expression. This times are measured in node 1 using one core for whole data set and five cores for clusters analysis and clusters extraction (one core computes SGDE for one individual cluster).

pluses ($\vec{\alpha}$) of the same size as the number of grid points. Then it looks for $|\alpha_{l,i}\phi_{l,i}| > threshold,\ \forall\ \alpha_{l,i}, \phi_{l,i} \in V_n^1$ (For notation see Section 2.2.1 on theory chapter) and chooses the highest values up to the number of refinement points (In our case it is 100 points). Finally, it refines those grid points that correspond to the selected values and make the new computation (with a bigger system of equations) using Conjugate Gradient method. This process is repeated with the new system of equations until it reaches the specified number of refinement steps(`ref_Steps`, which in our case is either 3 or 4). When doing this, the number of grid points increases, therefore we have a higher system to be solved by CG. Hence, the computation time increases.

In Figure 5.3 you can see the results in terms of Log-likelihood vs the number of grid points (after refinement). In this case the higher log-likelihood (equation 3.12) the better SGDE results are. In this plot all 12 scenarios for Gene Expression are presented. We see how in around 80% of the cases, SGDE analysis for the whole data set outperforms individual cluster analysis (each cluster is dimensionally reduced to a `q`-space then SGDE is computed). This effect could be due to the distortion added by PCA to two key characteristics: relative position with respect to other clusters and cluster radius. The former is completely lost and the latter increases up to the unit domain. This makes the SGDE computation using grids with Bounding Boxes lost its main advantage: The concentration of grid points in an smaller volume. To show this we plot Log-likelihood Vs Grid Point Density (GPD) in Figure 5.4.

On the other hand when `cluster extraction` is used, around 60% of the scenarios present better Log-Likelihood results than the whole data set analysis. In this case, it happends the contrary to what what is mentioned above when comparing whole data set

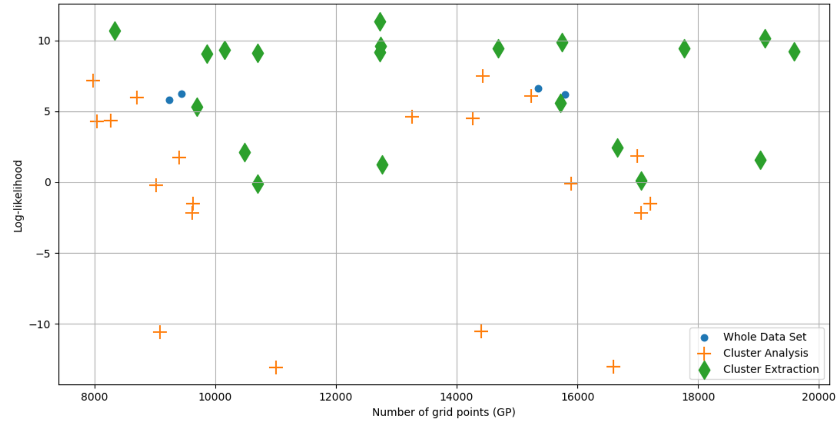


Figure 5.3.: log-likelihood vs Number of Grid points for Gene Expression. Dots (•) correspond to SGDE results for the whole data set, crosses (+) refers to cluster analysis results and diamonds (♢) when `cluster extraction` is used.

SGDE computation to individual cluster density estimation. In figure 5.4 we see that the higher is GPD the better Log-likelihood it tends to output. However, not being always the case.

To test SGDE results the input data set is divided in `Train` and `Test` subsets using *scikit-learn* library, with the function `model_selection.train_test_split`, with arguments `test_size=0.2` and `random_state=132`. The input sample can be either the whole data set or one cluster as mentioned before. This procedure is the same for Gene Expression as for Synthetic Data set.

5.4. Dimensionality Reduction and SGDE Results for Synthetic Data Set

Running the 12 numerical experiments for Synthetic data set we confirm that `mode=PCA` is effectively faster than hybrid mode. With PCA ranging from 0.19 to 0.28 seconds and hybrid taking between 3.67 to 5.05 seconds in terms of elapsed times. The elapsed time difference for Synthetic and Gene Expression data set lies in the original dimension of each data set, which is 10000 for the first one and 20531 for the second one. Therefore, it is expected to take more time to reduce dimensionality for Gene expression than for the Synthetic data set.

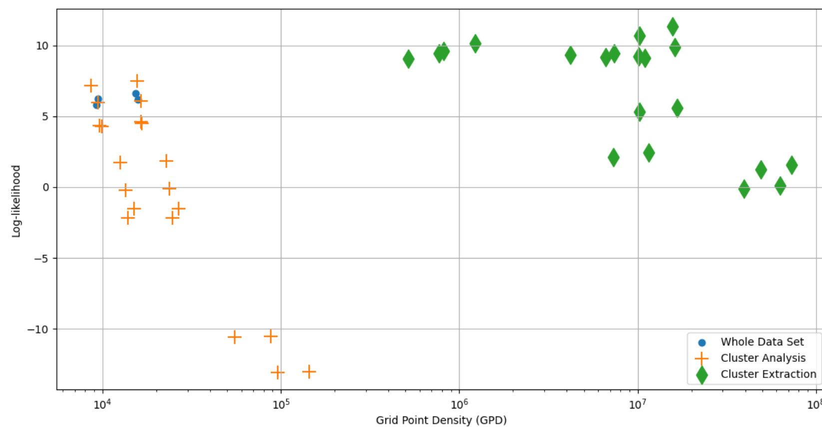


Figure 5.4.: log-likelihood vs Grid Point Density (GPD) (see Equation 3.11) for Gene Expression. It follows the same notation as Figure 5.3 and X-axis is in logarithmic scale. Here we effectively see that grids used in `cluster extraction` methodology has a higher GPD and it usually gives a better Log-Likelihood than the other two types of SGDE computations.

To compute SGDE for this data set we use one core for the whole data set and 3 cores for a parallel computations of each cluster, all of them in node 1. A similar analysis to what is done for elapsed times in Gene expression scenarios can be done for the Synthetic Data set.

In terms of log-likelihood we have similar results as Gene Expression; with `cluster extraction` results outperforming the other two. And whole data set SGDE computation having, most of the times, higher Log-likelihood than individual cluster analysis. This is shown in Figure 5.5. Furthermore, by plotting log-likelihood against GPD we see the tendency to have better results with higher GPD's (see Figure 5.6).

In Figure 5.5 we see that all 12 scenarios using `cluster extraction` form two vertical straight lines of diamonds ($\diamondsuit$) at 12252 GP and 18720 GP. It is a product of adaptive refinement algorithm which in this case are refined 3 and 4 times, respectively. Every step it refines the full set of 100 points in all scenarios. Despite that adaptive refinement algorithm usually gives a different number of grid points, in this case all 3 clusters of synthetic data set contains the same number of samples (300), encircled in a very similar cluster radius. Thus it results in the same output from adaptive refinement algorithm. It happens only in the case of `cluster extraction` because it mostly preservers the cluster radius and relative location of clusters from the original data set.

It is also important to highlight that by using `mode` either with PCA or hybrid does not

Computation Type	ref_Steps	Minimum Elapsed Time [s]	Maximum Elapsed Time [s]	Average Elapse Time [s]
Whole Data set	3	4320	6537	5428
(Parallel=False)	4	9456	10568	10012
Per Clusters Analysis	3	6092	9934	8070
(Parallel=True)	4	14358	15788	15105
Cluster Extraction	3	4116	7011	5533
(Parallel=True)	4	9804	11885	10864

Table 5.2.: Elapsed times of all scenarios for Synthetic Data Sets. This times are measured in node 1 using one core for whole data set and three cores for clusters analysis and clusters extraction.

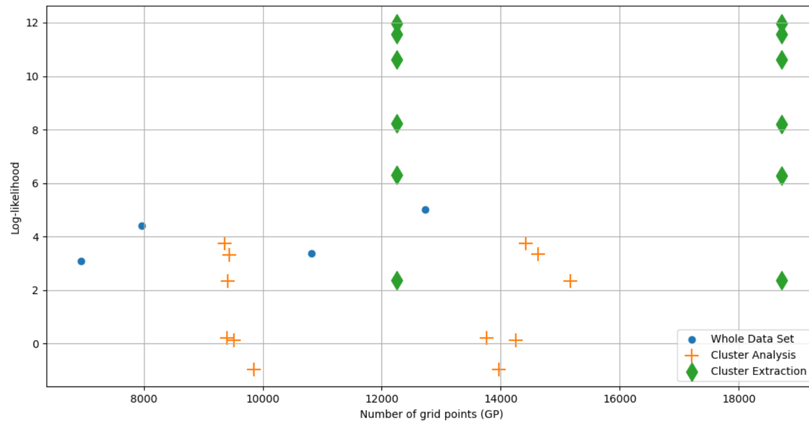


Figure 5.5.: log-likelihood vs Number of Grid points for Synthetic Data set. Dots (•) correspond to SGDE results for the whole data set, crosses (+) refers to cluster analysis results and diamonds (◇) when `cluster extraction` is used.

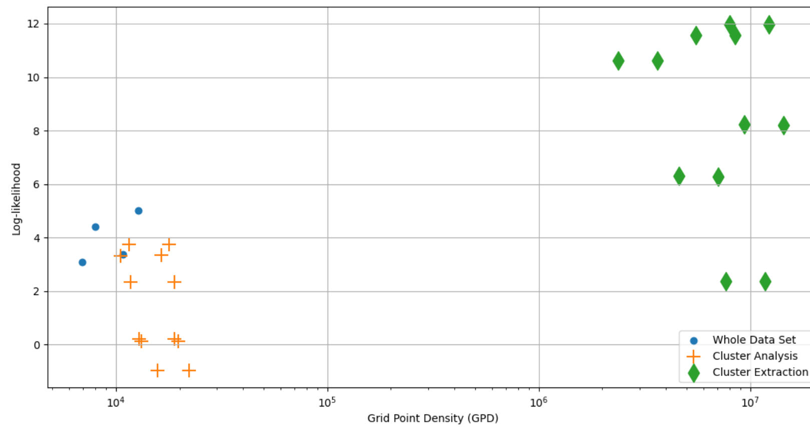


Figure 5.6.: log-likelihood vs Grid Point Density (GPD) (see Equation 3.11) for Synthetic data set. It follows the same notation as Figure 5.5 and X-axis is in logarithmic scale. Here we effectively see that `cluster extraction` methodology has significantly higher GPD and it usually outputs a better log-likelihood than the other two types of SGDE computations.

make any difference in terms of Log-likelihood for both of the data sets.

Another important remark is that despite the fact that LSH algotihm reaches a high Recall and Precision, it still could influence the results of individual cluster analysis and `cluster extraction` methodologies for Gene Expression. This is not the case for sysnthetic data set because LSH gets a 100% Recall and Precision and no missed samples.

6. Conclusions

Over the course of this thesis we have presented a pipeline that takes in a highly-dimensional clustered data set, preprocess it and perform two different type of computations to compare outputs. One is the conventional Sparse Grid Density Estimation using the whole data set and the other one subdivide the data set in subsets that correspond to its clusters. Various SGDE computations have been performed to both alternatives. The second approach is our proposal to estimate density using Sparse Grid in each individual cluster.

The two main algorithms have been successfully implemented into the pipeline; The first is a modified Locality Sensitive Hashing algorithm that uses Random Projections and Sketch matrices to hash samples into buckets of similar instances. And the second algorithm is a dimensionality reduction followed by an SGDE computation. Both algorithms have been validated using two sets of metrics, one for each algorithm.

The proposed approach to perform clustering using LSH proved to be highly competitive when tested on a real-world data set and a Synthetic one. Furthermore, the Sparse Grid Density Estimation, using grids with Bounding Boxes, applied on individual clusters achieved remarkable results in most of the different scenarios.

The presented pipeline can be further expanded by adding a module that instead of clustering the samples, it divides the domain into hypercubes that contains grids for density estimation and computes in parallel. Finally join the generated Probability Density Functions to have one single estimation to test. There exist another improvement which consist of taking the current pipeline and perform classification considering each cluster one individual class.

Bibliography

[1] Dimitris Achlioptas. Database-friendly random projections: Johnson-Lindenstrauss with binary coins. *Journal of Computer and System Sciences*, 66(4):671–687, 2003.

[2] Charu C. Aggarwal, Alexander Hinneburg, and Daniel A. Keim. On the surprising behavior of distance metrics in high dimensional space. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 1973:420–434, 2001.

[3] Charu C. Aggarwal, Alexander Hinneburg, and Daniel A. Keim. On the surprising behavior of distance metrics in high dimensional spaces. In *Proceedings of the 8th International Conference on Database Theory*, ICDT '01, page 420–434, Berlin, Heidelberg, 2001. Springer-Verlag.

[4] David Aha. Machine learning repository. https://archive.ics.uci.edu/ml/datasets/gene+expression+cancer+RNA-Seq. Accessed: 2020-10-25.

[5] Ethem Alpaydin. *Introduction to Machine Learning*. Adaptive Computation and Machine Learning. MIT Press, Cambridge, MA, 3 edition, 2014.

[6] Laura Anderlucci, Francesca Fortunato, and Angela Montanari. High-dimensional clustering via random projections, 2019.

[7] Richard Bellman. *Adaptive Control Processes: A Guided Tour*. 1961. First introduction of curse of dimensionality.

[8] Richard Bellman. *Dynamic Programming*. Princeton University Press, USA, 2010.

[9] Ella Bingham and Heikki Mannila. Random projection in dimensionality reduction: Applications to image and text data. In *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '01, page 245–250, New York, NY, USA, 2001. Association for Computing Machinery.

[10] Hans-Joachim Bungartz and Michael Griebel. Sparse grids. *Acta Numerica*, 13:147–269, 2004.

[11] Gerrit Buse. *Exploiting Many-Core Architectures for Dimensionally Adaptive Sparse Grids*. Dissertation, Technische Universität München, München, 2015.

[12] Sanjoy Dasgupta and Yoav Freund. Random projection trees and low dimensional manifolds. In *ACM Symposium on Theory of Computing (STOC)*, British Columbia, 2008.

[13] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the em algorithm. *JOURNAL OF THE ROYAL STATISTICAL SOCIETY, SERIES B*, 39(1):1–38, 1977.

[14] Jochen Garcke and Dirk Pflüger, editors. *Sparse Grids and Applications – Stuttgart 2014*. Springer, 2014.

[15] Jochen Garcke, Dirk Pflüger, Clayton G. Webster, and Guannan Zhang, editors. *Sparse Grids and Applications – Miami 2016*. Springer, 2016.

[16] Raymond Greenlaw and Sanpawat Kantabutra. Survey of clustering: Algorithms and applications. *Int. J. Inf. Retr. Res.*, 3(2):1–29, April 2013.

[17] Stephan Günnemann. Mining Massive Datasets Lecture: Scalability – Similarity Estimation, 2019.

[18] Markus Hegland, Giles Hooker, and Stephen Roberts. Finite element thin plate splines in density estimation. *ANZIAM Journal*, 42:712, 07 2009.

[19] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors. pages 604–613, 1998.

[20] William B. Johnson and Joram Lindenstrauss. Extensions of Lipschitz mappings into a Hilbert space. (September):189–206, 1984.

[21] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. Mining of Massive Datasets. *Mining of Massive Datasets*, 2014.

[22] Ping Li. Very sparse stable random projections for dimension reduction in lα (0 $<\alpha$ 2) norm. *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 440–449, 2007.

[23] Maximilian Luz. *Subspace-Optimal Data Mining on Spatially Adaptive Sparse Grids*. PhD thesis, University of Stuttgart, 2017.

[24] Jia Pan and Dinesh Manocha. Bi-level locality sensitive hashing for k-nearest neighbor computation. *Proceedings - International Conference on Data Engineering*, pages 378–389, 2012.

[25] Benjamin Peherstorfer. *Model Order Reduction of Parametrized Systems with Sparse Grid Learning Techniques*. PhD thesis, Lehrstuhl für Informatik mit Schwerpunkt Wissenschaftliches Rechnen, Technische Universität München, 2013.

[26] Benjamin Peherstorfer, Dirk Pflüge, and Hans-Joachim Bungartz. Density estimation with adaptive sparse grids for large data sets. *SIAM International Conference on Data Mining*, 2014.

[27] Dirk Pflüger. *Spatially Adaptive Sparse Grids for High-Dimensional Problems Spatially Adaptive Sparse Grids for High-Dimensional Problems Pflüger*. 2010.

[28] R. Redner and H. Walker. Mixture densities, maximum likelihood and the em algorithm. *Siam Review*, 26:195–239, 1984.

[29] John.A Richards. Clustering and unsupervised classification. In *Remote Sensing Digital Image Analysis*. Springer, 2013.

[30] scikit learn.org. Make blobs. https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_blobs.html. Accessed: 25.10.2020.

[31] scikit learn.org. Random projection. https://scikit-learn.org/stable/modules/random_projection.html. Accessed: 10.09.2020.

[32] sparsegrids.org. Sg++ github repository. https://github.com/SGpp/SGpp. Accessed: 2020-09-01.

[33] sparsegrids.org. Sg++ library. https://sgpp.sparsegrids.org/. Accessed: 2020-09-01.

[34] Ulrike von Luxburg. A tutorial on spectral clustering. *CoRR*, abs/0711.0189, 2007.

[35] Jingdong Wang, Heng Tao Shen, Jingkuan Song, and Jianqiu Ji. Hashing for Similarity Search: A Survey. pages 1–29, 2014.

[36] Zhipeng Wang and David W. Scott. Nonparametric density estimation for high-dimensional data—algorithms and applications. *Wiley Interdisciplinary Reviews: Computational Statistics*, 11(4):e1461, Apr 2019.

[37] Dongkuan Xu and Yingjie Tian. A comprehensive survey of clustering algorithms. *Annals of Data Science*, 2(2):165–193, 8 2015.

[38] Rui Xu and D. Wunsch. Survey of clustering algorithms. *Trans. Neur. Netw.*, 16(3):645–678, May 2005.