Parallel Self-Tuning Spectral Clustering on Apache Spark

by

Armand Grillet

Matriculation Number 376532

A thesis submitted to

Technische Universität Berlin
School IV - Electrical Engineering and Computer Science
Department of Telecommunication Systems
Service-centric Networking

Master Thesis

September 7, 2016

Supervised by:
Prof. Dr. Axel Küpper
Prof. Dr. Sebastian Möller

Assistant supervisors:
Boris Lorbeer, Dipl.-Ing.
Ana Kosareva, M.Sc.

Service-centric NETWORKING

Technische Universität Berlin

Eidestattliche Erklärung / Statutory Declaration

Hiermit versichere ich, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

I hereby declare that I have created this work completely on my own and used no other sources or tools than the ones listed.

Berlin, September 7, 2016 Armand Grillet

Abstract

This thesis proposes a new implementation of the self-tuning spectral clustering algorithm and a solution to use it on large datasets by parallelizing the computation. The algorithm studied has three qualities:

- It does not use an explicit model of data distribution (e.g. Gaussian) to find clusters of observations.
- The clusters in a dataset do not need to have the same density in order to be found by the algorithm.
- The algorithm does not require an input specifying the number of clusters in the dataset as it is self-tuned but only the minimum and maximum possible number of groups in it.

After describing the algorithm in details, an implementation developed in Scala is proposed. Compared to the two existing implementations, it is the first one to strictly follow the algorithm on steps such as the selection of the most probable number of clusters in the dataset.

Some steps of the algorithm are updated to make the computation faster. The resulting algorithm is usable as a library by other programs and different graphical user interfaces using it are presented.

The evaluation of the algorithm shows that the new implementation works as well as the one made for the original paper. The computation time of the algorithm is then evaluated for bigger datasets and shows that the algorithm is not usable in this configuration.

A solution to compute datasets containing a high number of small clusters is thus created. Using a k-d tree data structure, this thesis introduces a solution to cut datasets into tiles containing the same number of observations and process them in parallel using Apache Spark.

The parallel solution is evaluated and proved efficient, a dataset clustered in 5 hours by the original algorithm being clustered in 42 seconds. However, this solution only works on datasets that contain small clusters.

A solution for this problem, the tile border, is presented but future work could be done to make the parallelization usable on more various dataset types. Lastly, the computation time of the k-d tree and parallel computation is evaluated on datasets containing up to 10000 clusters.

Zusammenfassung

Diese Arbeit beschreibt eine neue Implementierung des “Self-Tuning Spectral Clustering”-Algorithmus und seine effiziente Anwendung auf großen Datensets durch Parallelisierung der Berechnungen. Der Algorithmus zeichnet sich durch drei Merkmale aus:

- Es wird kein explizites Modell der Datenverteilung (z.B. Gauß) verwendet, um Cluster zu finden.
- Die Cluster im Datenset müssen nicht die gleiche Dichte aufweisen, um vom Algorithmus erkannt zu werden.
- Die Anzahl der zu findenden Cluster muss nicht angegeben werden. Weil der Algorithmus self-tuned ist, genügt die Angabe der minimalen und maximalen Anzahl.

Nach der ausführlichen Erläuterung des Algorithmus, wird eine Implementierung in Scala vorgeschlagen. Im Vergleich mit den beiden existierenden Umsetzungen ist diese die erste, die jeden einzelnen Schritt des Algorithmus – wie z.B. die Auswahl der wahrscheinlichsten Anzahl an Clustern im Datenpool – chronologisch befolgt.

Manche Schritte des Algorithmus wurden angepasst, um die Rechenzeit zu reduzieren. Der entstandene Algorithmus kann als Programmbibliothek von anderen Programmen genutzt werden. Außerdem werden verschiedene grafische Benutzeroberflächen präsentiert.

Die Auswertung des Algorithmus zeigt, dass die neue Implementierung ebenso wie die des ursprünglichen Artikels funktioniert. Im Anschluss wird die Berechnungszeit des Algorithmus für größere Datensätze ausgewertet und festgestellt, dass der Algorithmus hier nicht anwendbar ist.

Aus diesem Grund wird anschließend eine Lösung für Datensätze mit einer hohen Anzahl von kleinen Clustern erstellt. Mithilfe einer k-d Baumstruktur wird in der vorliegenden Arbeit eine Lösung zur Teilung von Datensätzen in Subsets vorgestellt, die die gleiche Anzahl von Beobachtungen enthalten. Diese werden schließlich in Apache Spark parallel weiterverarbeitet.

Das Ergebnis dieser parallelen Berechnungen wird anschließend ausgewertet und die Effizienz der Lösung gezeigt. Der neue Algorithmus clustered einen Datensatz in 42 Sekunden, während der ursprüngliche Algorithmus für denselben Datensatz 5 Stunden benötigt. Jedoch funktioniert diese Lösung nur mit Datensätzen, die eine geringe Anzahl an Clustern enthalten.

Eine Lösung für dieses Problem, die “tile border”, wird vorgestellt, jedoch ist weitere Forschung notwendig, um die Parallelisierung für ein diverseres Feld an Datensätzen zu ermöglichen. Zum Abschluss wird die Berechnungszeit des k-d Baums und der parallelen Berechnungen bei Datensätzen mit über 10.000 Clustern ausgewertet.

Contents

1 Introduction

"Divide each difficulty into as many parts as is feasible and necessary to resolve it."

—René Descartes, Discourse on the Method

Teaching a computer how to recognize groups is a complicated task. Human beings can and do cluster data naturally but, to give this ability to computers, we need to develop machine learning and data mining algorithms.

Clustering is a common technique for data analysis used to group data objects into clusters, it is the core of this paper. This master thesis is about a clustering algorithm able to cluster groups in any shapes and requiring few inputs to work and how to make it work on large datasets.

1.1 Purpose of the thesis

Let us first understand the title of this thesis: *Parallel Self-Tuning Spectral Clustering on Apache Spark*. It is divided in two parts:

1. Self-Tuning Spectral Clustering (STSC), an algorithm created in 2005 by Lihi Zelnik-Manor and Pietro Perona [31].
2. Parallelization using Apache Spark, an engine to process data in a faster manner [18].

Another important element of the thesis is the use of a k-d tree [5] to cut a dataset into tiles to then cluster it in parallel, this concept does not depend on the clustering algorithm as it acts on top of it.

The motivation behind this thesis has been given by my supervisors:

> BMW produces cars taking pictures of road signs to show them in the head-up display of their new models. They want a solution to cluster these pictures to create a real-time map of the road signs in Europe. How to develop a clustering algorithm able to work on a billion observations, with new observations added all the time, in this configuration?

This thesis has been written with this use case in mind: how to work with a lot of observations? How to benefit from the fact that a road sign in Berlin has nothing to do with one in Paris?

1.2 Organisation of the paper

Following the template given by the Service-centric Networking research group, the chapters of this thesis are:

- A related work section that explains what are the main clustering algorithms, gives details about the spectral clustering algorithms and compare them to the self-tuning spectral clustering algorithm. The predecessors of Apache Spark are also described.
- A concept and design section dedicated to the steps of the self-tuning spectral clustering algorithm and the k-d tree used to parallelize the algorithm.
- An implementation section about the algorithms and applications developed for this thesis as well as the test environment used for the evaluation.
- An evaluation chapter to see if the new implementation works as the original one and a comparison of the sequential and parallel algorithms developed. The performances of the algorithms and the k-d tree computation are also tested depending on the size of the dataset clustered.
- A conclusion describing what has been accomplished and what could be done on top of this work.

1.3 Tools used during the development

The first decision has been to use the Scala programming language to create stsc, the library implementing the sequential and parallel algorithms[1]. This choice has been motivated by the Spark Application Programming Interface (API), available in Scala, and the quality of the libraries developed in Scala to do linear algebra. I had already programmed in Java, an interoperable language with Scala, before working on this thesis but it was the first time I was doing functional programming.

stsc uses extensively Breeze[2], a numerical processing library. The main advantages of this package are the methods copying what MATLAB can do with matrices and the global quality of the library in terms of computation time. This package is a major dependency of the Spark Machine Learning Library (MLlib [19]), a module of the Apache Spark project containing common learning algorithms and utilities.

One of the main element of this thesis is the implementation and evaluation of the sequential algorithm. A MATLAB implementation was attached to the original paper [31] and has been used to start the library presented in this thesis.

[1] `https://github.com/ArmandGrillet/stsc`
[2] `https://github.com/scalanlp/breeze`

2 Related Work

2.1 Different clustering models for different needs

Clustering is subjective, the same dataset can be partitioned differently depending on the application using it [13]. All the methods described in this chapter have advantages and drawbacks, compromises having to be done depending on what wants the user: a quality in the clustering or a good computation speed. Analyzing this related work shows the broadness of clustering, with many models to define what is a cluster.

2.1.1 The centroid model explained through *k*-means

k-means [17] is a clustering algorithm in three steps assigning a cluster to each observation of the dataset depending on its closest centroid [4]. The three steps are, following the Lloyd algorithm [15]:

1. k observations are randomly selected from the dataset as the initial means.
2. k clusters are created by associating every observation with its nearest mean.
3. the centroids of each of the k clusters becomes the new means.

The steps 2 and 3 are repeated until a maximum given number of iterations is reached or if we reach convergence, i.e. the step 2 does not update any cluster.

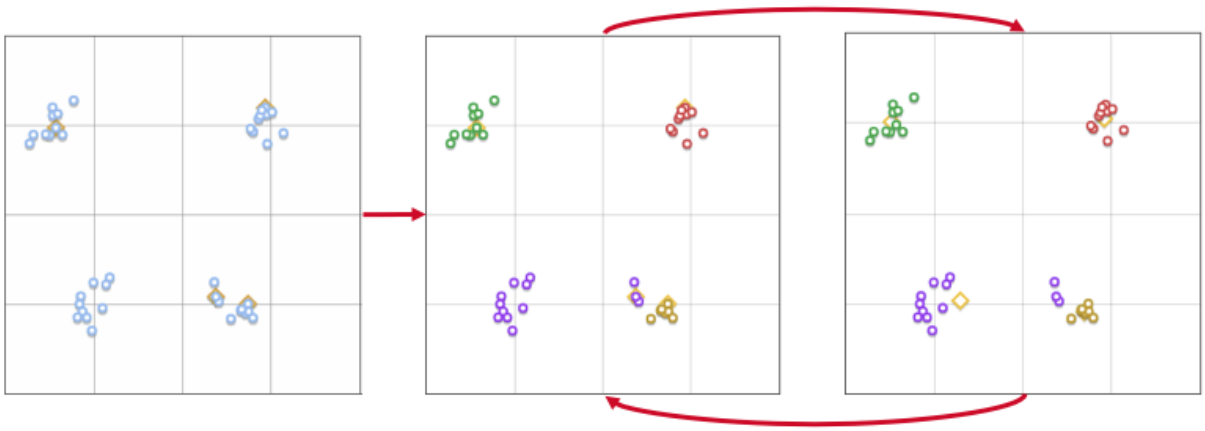

Figure 2.1: Visualization of the three steps of k-means with $k = 4$

The role of k-means is to minimize a squared error function $J = \sum_{j=1}^{k} \sum_{i=1}^{n} ||x_i^(j) - c_j||^2$ with c_j the centroid of the cluster j [15]. This concept of minimizing a cost function J is a fundamental part of STSC.

The algorithm has two limitations: it requires the number k of clusters in the dataset as an input parameter and its clustering model will produce equi-sized clusters privileging spherical clusters.

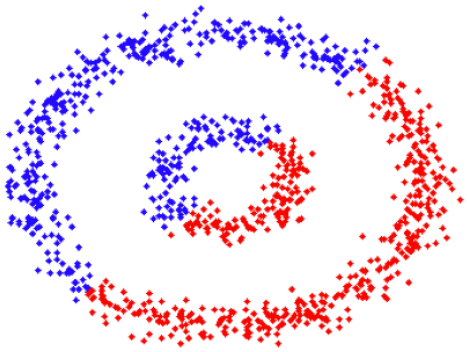

Figure 2.2: A badly clustered dataset due to the shape of the clusters, using k-means with $k = 2$ © Nick Alger

k-means, by defining clusters as a centroid surrounded by objects, offers a simplistic model that cannot handle noise in a dataset or non-globular clusters. Another problem is the fact that the result of the algorithm varies depending on its initialization [6], making the result of the clustering unstable.

2.1.2 The distribution model explained through Expectation-Maximization

Expectation-Maximization [21] (EM) is often described as a generalization of k-means. It is a soft clustering algorithm as it gives the strength of an association between an observation and every clusters instead of just assigning each observation to one cluster.

The EM algorithm for Gaussian mixtures starts by creating k randomly placed Gaussians $(\mu_1, \sigma_1), ..., (\mu_k, \sigma_k)$, we then compute for each observation the probability that it is in each cluster. In one dimension the likelihood that an observation i, in a dataset composed of n observations, belongs to a cluster k is:

$$E[Z_{ik}] = \frac{P(X = X_i | \mu = \mu_k)}{\sum_{i=1}^{k} P(X = X_i | \mu = \mu_k)}$$

This is the *Expectation* part, where we compute and normalize the probability that a data point i is in a cluster k. If we knew where are the μ of the clusters, having Z would be enough to know to which cluster is more likely to belong each observation. As we do not have this information, we need another step to compute the means of the clusters called the *Maximization*:

$$\mu_k = \frac{\sum_{i=1}^{n} E[Z_{ik}]X_i}{\sum_{i=1}^{n} E[Z_{ik}]}$$

We obtain the average of the X_i for k thus μ_k (again, this is in one dimension). Once we have the updated means, we compute the Expectation again and go back and forth between the two steps until convergence. The final result will give us the probabilities for each observation to be in each cluster thus more information than the output of k-means.

The EM algorithm uses a distribution model, i.e. clusters are defined as observations created by the same distribution, as the clusters are defined as k Gaussian distributions [30]. This assumption is useful if the dataset observations are known to follow this distribution but it misleads the algorithm if there is no mathematical model defining how the observations are positioned.

2.1.3 The density model explained through DBSCAN

DBSCAN [9] is the acronym of *Density-Based Spatial Clustering of Applications with Noise*. The algorithms takes as parameters ϵ, the maximum distance between an observation and the other points of the dataset used to create the ϵ-neighborhood, and $minPts$, the minimum number of points required to form a cluster.

We start by randomly selecting an observation p that has not been classified and compute its ϵ-neighborhood $N\epsilon(p) = q \in D|dist(p,q) \leq \epsilon$. If $N\epsilon(p) \geq minPts$ a cluster is started, otherwise the observation is classified as noise.

To find all the observations in a cluster, we take all the data points within $N\epsilon(p)$, expand the cluster by checking their ϵ-neighborhood and add the unassigned observations in the neighborhood to the cluster. Thus, starting with one point p, we get all the observations directly reachable but also the points that are reachable through a chain within the cluster.

The main advantage of DBSCAN compared to the previously described solutions is that we do not need to know the number of clusters to do the clustering. But the algorithm is highly dependent of ϵ, making it inefficient when clustering a dataset composed of clusters with different densities.

A clustering algorithm using the same model has been developed to fix the density issue, OPTICS [2]. It is based on the same computation as DBSCAN but it adds two notions:

Core distance If $N\epsilon(p) \geq minPts$, it is the Euclidean distance between p and its $minPts^{th}$ nearest observation.

Reachability distance If $N\epsilon(p) \geq minPts$, it is $max(dist(p,q), coreDist_{\epsilon,minPts}(p))$ with q being another observation in the dataset.

Using these two notions we create a reachability-plot with the ordering of the points on the x-axis and the reachability distance on the y-axis. We obtain obtain a bar chart where each object's reachability distance in the order the object was processed thus where the clusters can be found as valleys.

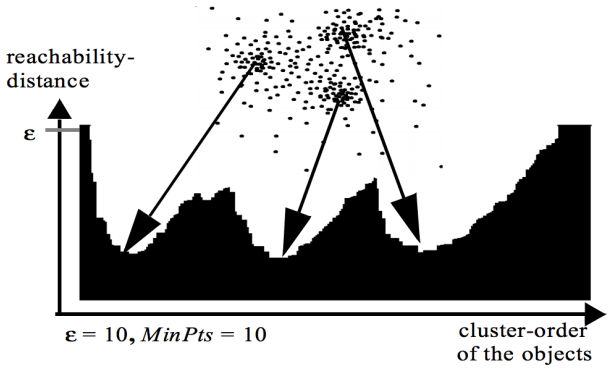

Figure 2.3: A reachability-plot obtained from a dataset composed of three clusters © Mihael Ankerst et al.

The core distance concept solves the density problem but increases the complexity of the algorithm and does not solve the main issue with the density model: we need a density drop between the clusters to see them.

Contrary to the EM algorithm, DBSCAN and OPTICS are unable to correctly cluster datasets composed of overlapping Gaussian distributions. However, by defining clusters as zones of high density, the model works well with any kind of cluster shapes and the noise in a dataset is correctly detected.

2.1.4 Conclusion

Many models used for clustering models exist, all with different concepts, advantages and drawbacks. We have seen how work *k*-means, EM, DBSCAN and OPTICS and see the differences between hard and soft clustering. All these algorithms have many variations made to solve specific problems that we will not see in this paper.

There are many factors when using a clustering algorithm: the size of the dataset and its dimensionality, the number of clusters and their shape [1]. The big-O complexity of the algorithms also vary depending on their computation, a factor of importance for real-time applications.

The main subject of interest in this thesis is spectral clustering, the method used in the algorithm developed. We will learn more about it before explaining the concept and design of the STSC algorithm.

2.2 Spectral clustering

Spectral clustering is based on the analyze of the spectrum of Laplacian matrices [16]. This computation does not require an explicit model of data distribution (e.g. Gaussian) thus it works with clusters that have unknown shapes.

It uses concepts from linear algebra, namely eigenvectors and eigenvalues, to do linear transformations on matrices. Before analyzing what are the spectral clustering algorithms related to STSC, here are the common steps of spectral clustering:

1. The creation of a similarity matrix (also called affinity matrix) A from the projection of the observations into $\mathbb{R}^n$. Each row defines the affinities of an observation compared to

the other observations of the dataset thus A is symmetric and hollow.

2. Normalization of the affinity matrix to get the graph Laplacian L using A and the Degree matrix, a diagonal matrix with $D_{i,i} = \sum_{j=1}^{n} A_{ij}$ which is the average of A. Diverse Laplacian exist depending on the algorithm: unnormalized, normalized, generalized, relaxed. Some Laplacian will scale the rows of A, some the columns, it all depends on what needs to be need with l in the next steps.
3. We compute the first k eigenvectors $u_1, ..., u_k$ solving an Eigenvalue problem such as $Lu = \lambda Du$
4. We create a matrix containing $u_1, ..., u_k$ as columns and cluster the rows in that subspace.

The third step uses many concepts from linear algebra that we need to define to fully understand the idea of spectral clustering.

2.2.1 Eigenvectors and eigenvalues

Let us use a matrix to explain these two concepts:

$$A = \begin{pmatrix} 4 & 3 \\ 1 & 2 \end{pmatrix}$$

Almost all vectors change direction when we apply the linear transformation represented by A. The vectors x's that are in the same direction as Ax are rare and called eigenvectors [27].

To obtain these eigenvectors, we first compute the eigenvalues of the matrix solving the basic equation $Ax = \lambda x$. To obtain them, we look at $det(A - \lambda I)$:

$$\begin{vmatrix} 4-\lambda & 3 \\ 1 & 2-\lambda \end{vmatrix} = \lambda^2 - 6\lambda + 5 = (\lambda - 5)(\lambda - 1)$$

We now have the two eigenvalues of A: 5 and 1. We can now compute the eigenvectors.

For $\lambda_1 = 5$ we have $Ax_1 = 5x_1 \Leftrightarrow (A - 5I)x_1 = 0$ thus we need to solve this system of equations:

$$\begin{cases} -x + 3y = 0 \\ x - 3y = 0 \end{cases} \Rightarrow x_1 = \begin{pmatrix} 3 \\ 1 \end{pmatrix} \quad \text{and} \quad Ax_1 = \begin{pmatrix} 4 & 3 \\ 1 & 2 \end{pmatrix} \begin{pmatrix} 3 \\ 1 \end{pmatrix} \Leftrightarrow Ax_1 = \begin{pmatrix} 15 \\ 5 \end{pmatrix} \Leftrightarrow Ax_1 = 5x_1$$

For $\lambda_2 = 1$ we have $Ax_1 = x_1 \Leftrightarrow (A - I)x_1 = 0$ thus we need to solve this system of equations:

$$\begin{cases} 3x + 3y = 0 \\ x + y = 0 \end{cases} \Rightarrow x_2 = \begin{pmatrix} 1 \\ -1 \end{pmatrix} \quad \text{and} \quad Ax_2 = \begin{pmatrix} 4 & 3 \\ 1 & 2 \end{pmatrix} \begin{pmatrix} 1 \\ -1 \end{pmatrix} \Leftrightarrow Ax_2 = \begin{pmatrix} 1 \\ -1 \end{pmatrix} \Leftrightarrow Ax_2 = x_2$$

Eigenvectors of matrices derived from the dataset to cluster are what use spectral clustering algorithms [23].

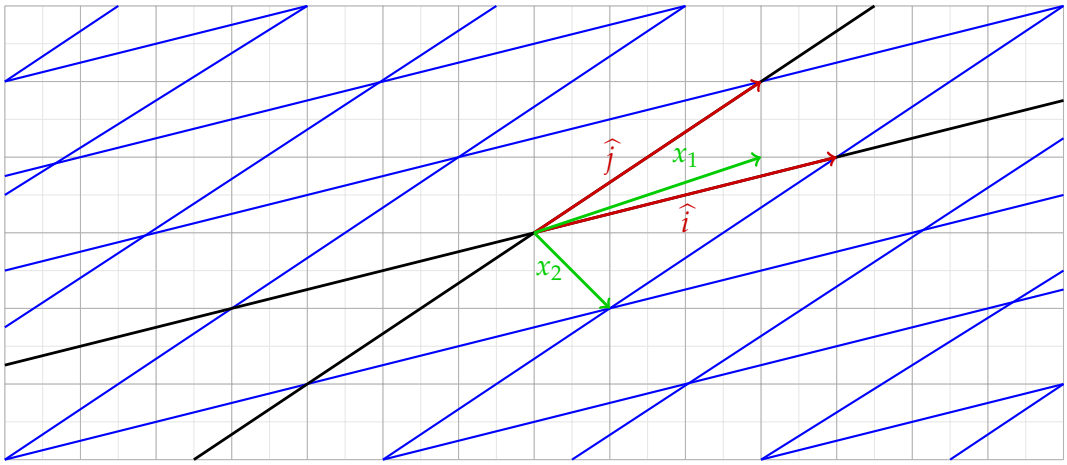

Figure 2.4: A and its two eigenvectors.

2.2.2 Algorithms related to STSC

2.2.2.1 Unnormalized spectral clustering

The first step of this algorithm is to create a similarity graph to represent the local neighborhood of each observation from the dataset [20], different types of graph exist [16]:

k-nearest neighbor graph Connects an observation to its k neighbors, the result is not symmetric thus the result is a directed graph. To make it undirected we either ignore the directions of the edges or we only do the connection (also undirected) between two observations i and j if j is one of the k^{th} nearest neighbor of i and vice versa.

Fully connected graph Connects all the observations and give a weight to the connection using a similarity function.

ϵ-neighborhood graph Connect data points i and j if the Euclidean distance between them is inferior to ϵ.

We then create a weighted adjacency matrix W from the graph and compute the laplacian $L = D - W$. We compute the first k eigenvectors of L (§2.2.1) and put them as columns in a matrix U. We then use the k-means algorithm to cluster U by considering each row as a vertex.

The results can then be applied on the original matrix, i.e. if the first row of U has its biggest value is in the second column of U then the first observation of the dataset belong to the second cluster.

This computation is the most basic one in spectral clustering, all the algorithms used are straightforward as it is not normalized and the clustering algorithm applied on U is k-means (§2.1.1).

2.2.2.2 Normalized spectral clustering by Shy and Malik

The algorithm described is the one invented by Shy and Malik [25]. The only difference compared to the unnormalized algorithm is that we compute the first k generalized eigenvectors

solving the eigenproblem $Lx = \lambda Dx$ instead of the eigenvectors solving the standard eigenproblem $Lx = \lambda x$.

This computation of the generalized eigenvectors of the laplacian $L = D - W$ is equivalent to the computation of the eigenvectors of the normalized graph Laplacian $L_{rw} = D^{-1}L$ thus the algorithm is normalized [7].

2.2.2.3 Normalized spectral clustering by Ng, Jordan, and Weiss

The algorithm described is the NJW algorithm invented by Ng, Jordan, and Weiss [23], it is the main related work for STSC (described in the part §1.1 of the original paper [31]).

The creation of the similarity graph is the same as for the unnormalized algorithm but the Laplacian created next is $L_{sym} = D^{-1/2}LD^{-1/2}$. As before, the Laplacian has n non-negative real-valued eigenvalues.

We then compute the first k eigenvectors of L_{sym} and create the matrix U that we normalize to the unit length by creating a matrix T such as $T_{ij} = U_{ij}/(\sum_{k=1}^{j} U_{ik}^2)^2$.

This second normalization step is due to the fact that if the degrees of the vertices of L_{sym} differ a lot or have a very low degree, the corresponding entries in U are very small. To cluster correctly the eigenvectors we need a matrix with one non-zero entry per row, this is why we compute T. We then apply k-means by interpreting each row of T as an observation.

The three algorithms described only vary in the computation of the Laplacian but they can also vary depending on the similarity graph used.

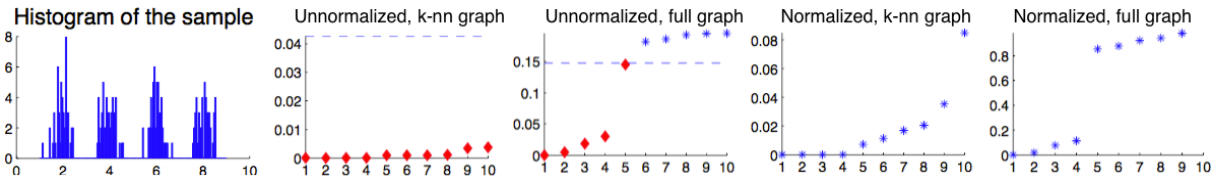

Figure 2.5: Dataset clustered on the left, 10 first eigenvalues of the Laplacian obtained from four spectral clustering algorithms. From left to right: unnormalized with k-nearest neighbor and fully connected graph, normalized with k-nearest neighbor and fully connected graph © Ulrike von Luxburg

2.2.3 Conclusion

We have seen the mathematical concepts around spectral clustering algorithms related to STSC. Spectral clustering offer advantages compared to k-means and Expectation-Maximization by working with any types of cluster shapes. However, the algorithms we have reviewed still request as inputs a number k of clusters to construct.

This is the main point that STSC is solving compared to other spectral clustering solutions, as we will see in the next chapter. By detecting the number of clusters in a dataset, we can offer an algorithm that only request a dataset and a maximum number of potential clusters to work, an advantage for numerous applications.

2.3 Parallel processing

The goal of this thesis is also to parallelize STSC using Apache Spark. This library inherit from related solutions developed previously to easily process data on large clusters.

2.3.1 Google MapReduce

MapReduce is a programming model and an implementation (originally made by Google) to process data in parallel on a cluster [8]. It handles the risk of large distributed workloads lost due to disk failures by using the Google File System [10] thus it is a great solution to do batch processing on commodity hardware.

The programming model offers two methods:

Map() Applies an elemental function on all element of a sequence separately, in parallel. The output can be of another type than the input, the goal is to transform or filter all the elements of a given dataset.

Reduce() Reduces the sequence returned by $map()$ by applying a combining function such as an aggregate.

This model brings functional programming to big data processing by allowing data scientists to use these two common methods in a cluster without having to handle the communications between each machine in it (this part being handled by the MapReduce implementation).

The solution being proprietary, open-source developers have reproduced the file system and programming model in a new framework: Apache Hadoop.

2.3.2 Apache Hadoop

Hadoop is composed of a file system (HDFS [26]), a cluster scheduler (YARN [28]) and a MapReduce implementation. Spark uses HDFS as a data source and can run in a YARN environment.

The Google and Apache MapReduce implementations offer a great solution for batch processing but several use cases need something else (e.g. for analyzing data streams). Developers built specialized systems on top of MapReduce to do what they wanted but they were only workarounds, this is why Spark exists and offers a solution compatible with many use cases of data science.

2.3.3 Apache Spark

Spark offers a unified engine for enterprise data workflows to do batch processing but also complex queries on distributed datasets (Spark SQL [3]), fault-tolerant streaming applications (Spark Streaming), machine learning (through the algorithms in MLlib [19]) and graph processing (GraphX [29]).

Spark has its own implementation of MapReduce, it adds an abstraction, resilient distributed datasets (RDDs), to improve the computation time of iterative machine learning jobs [18].

3 Concept and Design

3.1 The self-tuning spectral clustering algorithm

The self-tuning spectral clustering algorithm (STSC) is divided into multiple steps [31]. We will apply in this part the original computation, as done in the MATLAB code given with the paper, on a dataset composed of 17 observations:

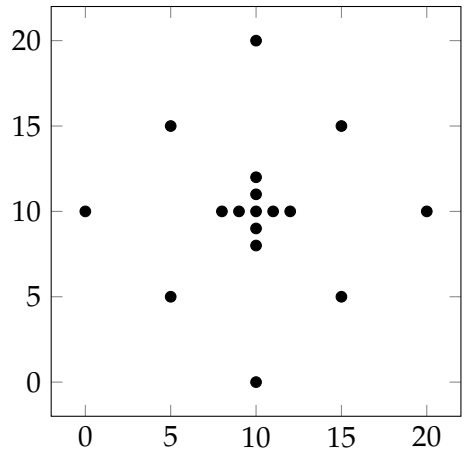

Figure 3.1: The dataset computed step by step, composed of two datasets.

There are the two clusters: one cross in the middle of the graph and a less dense cluster surrounding it. We give to STSC three inputs: the dataset, a minimum and a maximum number of possible clusters.

The only strict limit concerning the inputs is the minimum number of possible clusters that cannot be below 2 due to one step of the algorithm (§3.1.5).

3.1.1 The local scales

The local scale σ is a parameter defining how fast the affinity between two observations decreases as the distance between the two increases. Instead of having only one σ for the entire dataset, we compute a local scale σ_i for each data point s_i. The local scales are computed using the formula:

$$\sigma_i = d(s_i, s_K)$$

s_K is defined in the original paper as the 7^{th} nearest neighbor of s_i because "we used a single value of $K = 7$, which gave good results" [31]. To get this information, we first compute a matrix of the Euclidean distances between each data point to find the nearest neighbors.

The purpose of having one local scale per observation is to handle datasets containing clusters that have different densities. By having different scaling parameters, we can tune the affinity between two observations depending of their surroundings.

In our example, the observation $s_i = (10, 0)$ has a rounded local scale of 10 whereas the observation $s_j = (10, 10)$ has a local scale of 2. This value will impact the locally scaled affinity matrix.

3.1.2 The locally scaled affinity matrix

At its core, a clustering algorithm has to cluster observations that are similar and separate the ones that are not. In practice, we need to find the affinity between each observation in its context, i.e. the dataset. Using the local scales, we can create a locally scaled affinity matrix $\hat{A}$:

$$\hat{A}_{ij} = \exp\left(\frac{-d^2(s_i, s_j)}{\sigma_i \sigma_j}\right)$$

Let us see what is the affinity between the two previous observations of our example, $s_i = (10, 0)$ and $s_j = (10, 10)$ and $s_k = (0, 10)$.

$$\hat{A}_{ik} \simeq 0.138 \quad \text{and} \quad \hat{A}_{jk} \simeq 0.00690 \quad \text{and} \quad \hat{A}_{jk} = \hat{A}_{ij}$$

The Euclidean distances are the same between the three data points but the affinity $\hat{A}_{jk}$ is 20 times less than $\hat{A}_{ik}$. This is due to the local scales and the position of j, an observation close to eight other data points.

3.1.3 The normalized affinity matrix

Once we have a locally scaled affinity matrix, we normalize it to standardize the affinities in order to speed up the next steps that will involve cost decay and an optimization through Givens rotations.

We first compute the sum of the affinities for each observation to obtain the diagonal matrix D. Once we have it, we create the normalized affinity matrix $L = D^{-1/2}\hat{A}D^{-1/2}$.

3.1.4 The largest eigenvectors

Spectral clustering algorithms, by definition, use the eigenvalues and eigenvectors (i.e. the spectrum) of the affinity matrix to cluster a dataset. STSC uses the C largest eigenvectors of L (C being the maximum possible number of clusters in the dataset given as an input) in the next steps of the algorithm.

We obtain the largest eigenvectors by doing a standard value decomposition [14]. They are then stored in a matrix X that will be used in the next steps where the first column of X is the

biggest eigenvector and so on.

3.1.5 The best rotations

One of the task of STSC is to get a cost for each possible number of clusters in the dataset between the minimum and the maximum values given by the user to find the most probable number of groups. This computation is done incrementally.

We start with the minimum number of possible clusters *min* by taking the *min* first columns of X, we rotate them by applying a Givens rotation in an stochastic gradient descent scheme [11] to find the optimal rotation R. The point of comparison when doing the rotation is the minimization of the cost function J:

$$J = \sum_{i=1}^{n} \sum_{j=1}^{C} \frac{Z_{ij}^2}{M_i^2} \quad \text{with} \quad Z = XR \quad \text{and} \quad M_i = \max_j Z_{ij}$$

We update the parameters in the opposite direction of the gradient ∇J following a learning rate, also called the step size. The reason behind J is to find the best alignment with the canonical system for X, where each row of XR has only one value that stands out (the other values need to be near 0).

This computation has one problem: it is not possible to get an interesting cost if we rotate only one eigenvector (i.e. we try to find the cost of having only one cluster in the dataset). In that case, $J = 1$ as $\sum_{j=1}^{C} Z_{ij} = Z_{i1}$ and $Z_{i1} = M_i$ thus J will be at its minimum and *cBest* will be 1 even if other cluster numbers are more suitable for the dataset.

This is why the implementation made for the original paper verifies if users try to check if a dataset is one cluster and does not compute this case, without raising an issue nor having a comment about this limitation in the original paper.

To apply the gradient descent, we need to minimize J over $\Theta \in [-\pi/2, \pi/2)^K$ with $K = N(N-1)/2$ [11]. Each Θ_k is thus a value θ representing the angle of the Givens rotation of the matrix in the coordinate plan (i, j).

The list of $(i, j) \in 1, 2, ..., C^2$ is fixed, defined by $(i < j)$ and created when minimizing the aligning rotation for a given X. The main problem is that each Θ_k is dependent of the others thus finding the best rotation has a time complexity of $O(2^K)$. We will see the consequences of this exponential complexity in the evaluation chapter (§5).

Once the first rotation for *min* is optimal, we save its cost and the associated rotation XR for the next steps and we horizontally concatenate XR to the $(min + 1)^{th}$ eigenvector, corresponding to the $(mi + 1)n^{th}$ column of X. By using the alignment of the previous eigenvectors we make the computation of the new alignments faster as the initialization is already good. This process is the incremental gradient descent scheme.

One important note: even if the paper only speak about a computation of the cost, the original computation transform it in a value called quality $Q = 1 - \frac{(J/n)-1}{C}$. This value ranges between 0 and 1, 1 being the best possible quality.

Number of eigenvectors to rotate	Quality (rounded)
2	0.9956
3	0.9533
4	0.9245
5	0.9434
6	0.9344

Table 3.1: Quality of the Givens rotation depending on the number of eigenvectors (i.e. columns) rotated for the dataset §3.1.

3.1.6 Selecting the rotation

We can see in the table §3.1 that the algorithm correctly found that there are two clusters in the dataset §3.1 as the biggest rounded quality is linked to 2 clusters. One detail concerning the selection of the best number of clusters is that "if several group numbers yield practically the same minimal cost, the largest of those is selected" [31].

We do the opposite when comparing qualities: the maximal quality is the one selected if two are nearly the same. The largest number of clusters is selected if the difference is up to 0.001% apart when comparing the costs in the paper and up to 0.001 when comparing the qualities in the original implementation.

3.1.7 Clustering the observations

Now that we know how many clusters are in the dataset, we want to know in which cluster is each observation. To do this step we take the optimal rotation RX for the C_{best} number of clusters.

We then assign the data point i (each one is a row in RX) depending on the biggest value in the corresponding row, thus the cluster where belongs i is $max_j(Z_{ij}^2)$.

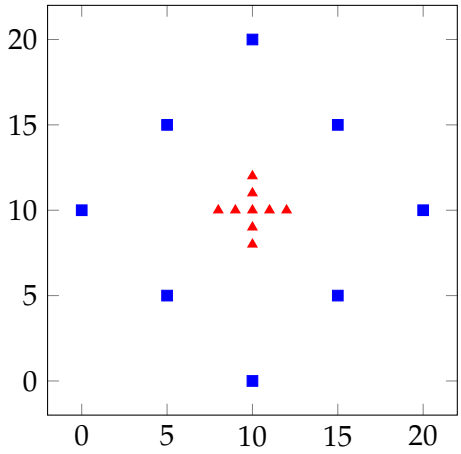

Figure 3.2: The two clusters obtained from the algorithm when applied on §3.1. We can see that the algorithm clustered the dataset correctly, even with the differences of density between the two clusters.

3.2 The *k*-d Tree

3.2.1 Definition

A *k*-d tree [5] is a binary tree that stores a set of points from a k dimensional space [22]. For our thesis, we use a *k*-d tree having as nodes an element composed of a tile (i.e. the coordinates defining the node) and two children nodes. If the node has no children, it is a leaf.

We have seen in the introduction that we want to cluster a big amount of observations representing road sign. Data points representing a road sign in Berlin have no link to observations concerning a road sign in Paris thus we can use a *k*-d tree to cut the dataset into multiple tiles that can be processed separately. This concept can be broadened to all datasets containing many small clusters, the tree being independent of the clustering algorithm used.

Here is an example of a simple dataset on which we will apply a *k*-d tree:

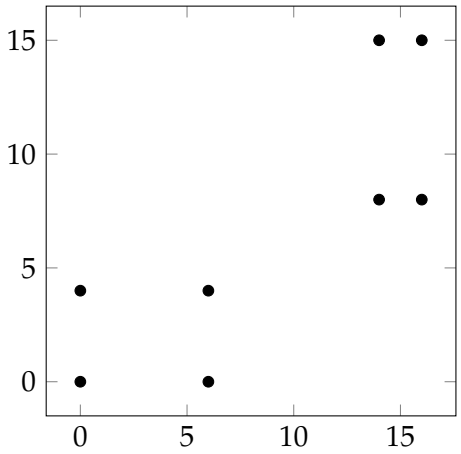

Figure 3.3: Dataset used to compute a *k*-d tree in this section.

The tile contained in the node is composed of two vectors *mins* and *maxs* containing the coordinates of the node. For example, a Tile((0, 5), (10, 15)) is a rectangle with $x_{min} = 0$, $x_{max} = 10$, $y_{min} = 5$ and $y_{max} = 15$.

To create a *k*-d tree, we need:

- A dataset to know where to place the tiles.
- The maximum number of observations per tile or the expected number of leafs in the tree, this parameter is used to know when to stop the tessellation.
- A border width, a parameter that is explained at the end of this section (§3.2.3).
- A cut function used to know how to compute the tile of the two children nodes of a parent.

The cut function can use many parameters such as the dimensions of the parent's tile or or the observations inside it. In two dimensions, a cut function will find if we need to cut the parent tile vertically (1^{st} dimension) or horizontally ($2nd$ dimension) then compute the median of the observations in the parent tile in the dimension selected to know the tiles of the children.

3.2.2 Representations

Let us use the dataset §3.3, a maximum number of observation of 2, a border width of 0 and a cut function using the size of the parent tile to know if we need to cut vertically or horizontally to create the children tile. If the tile is square we cut it vertically as it is the dimension 1.

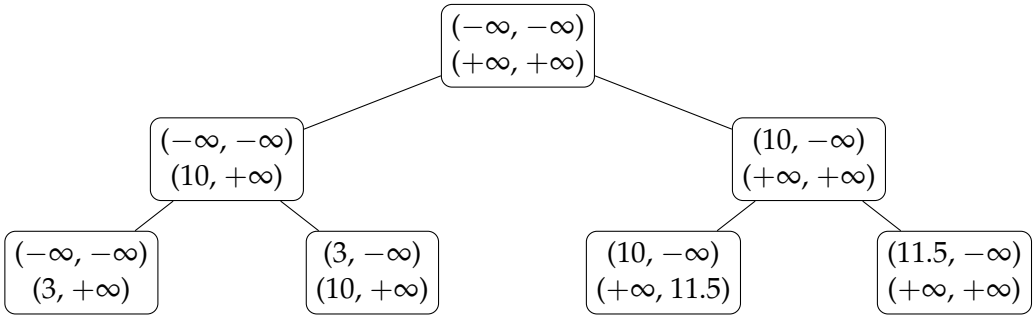

Figure 3.4: *k*-d tree and nodes' tiles coordinates.

The tree can be visualized through a tessellation of the observations in a 2D space:

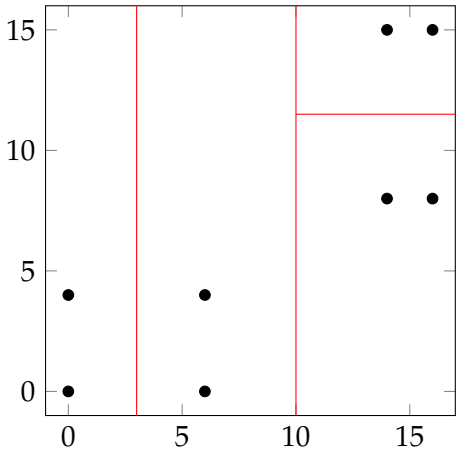

Figure 3.5: *k*-d tree and nodes' tiles coordinates.

The graph only shows a small portion of the two-dimensional space where are visible the cuts between the tiles. As we can see in the tree we always set the root of the tree to be a tile with infinite *mins* and *maxs* to cover any possible observation.

To know what will be the coordinates of the children of a node, we first get on which dimension will the parent node be cut and then compute the mean of the observations in the parent tile in the dimension that interest us. We use the mean to create two children tiles that will have the same number of observations and to get a tree where all the nodes at the same level are surfaces containing the same number of observations.

Using a *k*-d tree is a solution to parallelize the workload when clustering a dataset by giving one tile to one machine but it is costly as we need to compute the mean of the observations in a tile when we create children. We will evaluate and compare the speed of the computation using STSC and a *k*-d tree compared to just STSC in the evaluation chapter.

3.2.3 The border width

An important parameter of the *k*-d tree is its border width. One of the main problem when cutting a dataset is to also cut the clusters composing it, the border width is the solution created for this issue.

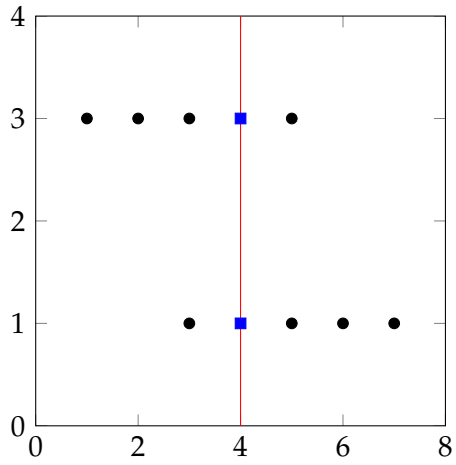

Figure 3.6: The *k*-d tree in this dataset cuts the clusters. The two observations represented as blue squares belong to the tile with the coordinates $((4, -\infty), (-\infty, +\infty))$

The solution to fix the clustering problem is the border width input, this parameter is used by the algorithm to take all the observations in a tile but also the ones contained in the outer limits of an area limited by this border when clustering the data points of a tile.

To cancel duplicates, we compute the center of the clusters obtained in each tile and if it is outside the strict border of the tile (without the border width), we remove the cluster and the observations forming it. This solution removes the duplicates that are introduced by having the same observation computed multiple times.

This solution has to be placed in the context of the thesis, with observations forming small clusters in a dataset and tiles covering thousands of observations. In that case the border is minimal compared to the surface of the tile.

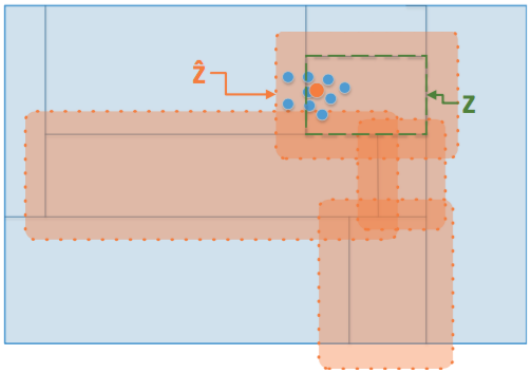

Figure 3.7: A tile Z and the same tile with its border $\hat{Z}$, © A. Kosareva and B. Lorbeer

The border width as one issue: how to select it? As its role is to solve problems when finding a cluster that is between two tiles, its size should be the maximum length of a cluster in the dataset divided by 2.

Knowing the maximum length of a cluster in the dataset would mean knowing the clusters of the dataset we want to cluster. To solve this problem, the easiest solution would be to select a large border width, but how large?

This problem also highlights the issue concerning the computation of the *k*-d tree: how many leafs to create in a dataset? If STSC is fast when computing a large number of clusters, the dataset will not need to be cut in a high number of leafs and the border width can be set to a safe value without impacting the computation time of the algorithm.

Some of these questions have an answer in the evaluation chapter, but the lack of a scientific method to cut the dataset is a problem that has not been resolved.

4 Implementation

The first part of the implementation has been to take the code given with the original paper [24] and transform it in a more suitable language for parallelization. Apart from the original MATLAB code, the implementation is inspired by the only other implementation of the algorithm available [1].

Both of these implementations do not offer an Application Programming Interface (API) to use the algorithm simply. The original implementation has a tangled control structure making it hard to understand the different steps. It also mixes the algorithm, tests, resources and computations made to compare the algorithm with other solutions like *k*-means in a same folder containing all the code made to write the algorithm.

The main concepts used by the algorithms such as vectors, matrices, singular value decomposition and euclidean distances have directly been taken from the Breeze library[2].

4.1 Differentiating the paper and the original code

The MATLAB code given with the paper does not strictly follow the original computation. There are three main differences:

1. The paper compares costs to select the best possible number of clusters wheras the MATLAB code uses a value $quality = 1 - \frac{(cost/X_{rows})-1}{X_{cols}}$.
2. When selecting a group number, it is said in the paper that the largest group number providing the minimal cost is chosen with costs up to 0.01% apart considered the same. The selection in the original code is made with qualities up to 0.001 apart considered the same, thus with a differentiation of 0.1%.
3. The best aligning rotation is found using a true derivative in the original paper. The code offers two methods called *numericalDerivative* and *trueDerivative* to find it and the numerical derivative is used by default.

To choose what was the best way to handle these differences, the solution has been to test the algorithm on the datasets given as resources in the original code (§4.1) and used in the paper.

As the original code was in MATLAB and the second application was not able to take something else than datasets of one dimension as inputs, it has not been possible to compare the computation of these solutions with the new one. The comparison has thus been made between the results presented in the original paper [31] and the ones obtained when using the

[1] `https://github.com/pthimon/clustering`
[2] `https://github.com/scalanlp/breeze`

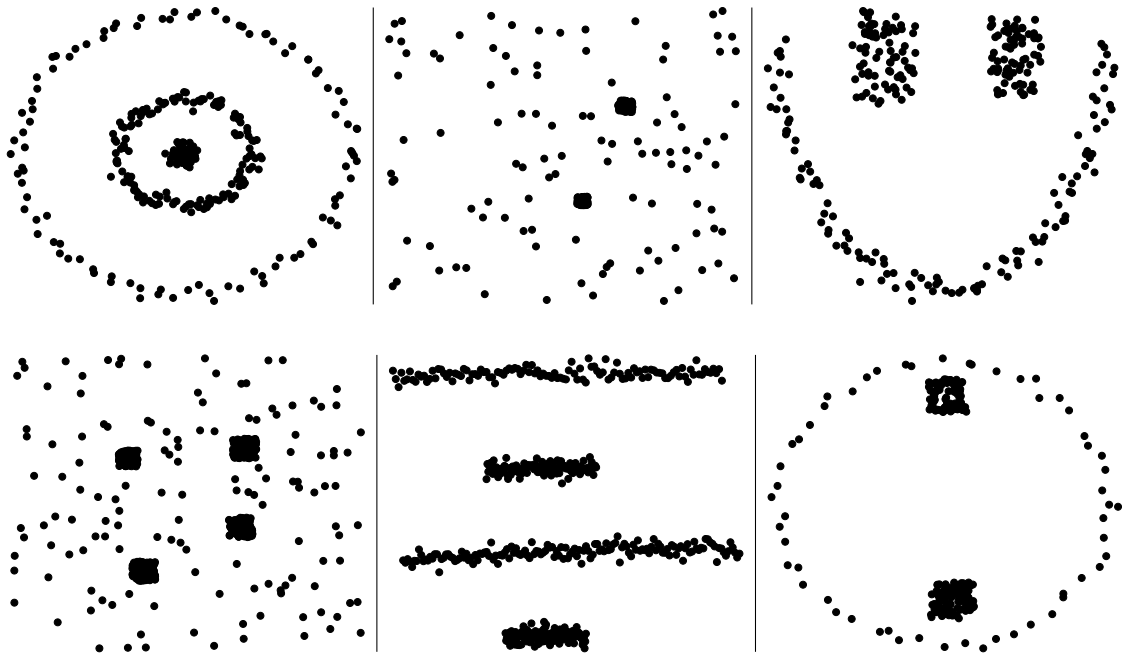

Figure 4.1: The six datasets given with the original code and used in the paper, represented in a R^2 space.

new algorithm.

4.2 Choosing the derivative

As we have seen in the Concept and Design chapter, the most complex part of the algorithm is the recovery of the best aligning rotation (§3.1.5). This requires to optimize the matrix of eigenvectors by applying a Givens rotation that minimize the cost function.

To find the best Θ to rotate the matrix, we need to minimize the cost function J. This minimization has been implemented two times in the original implementation using a numerical derivative and a true derivative.

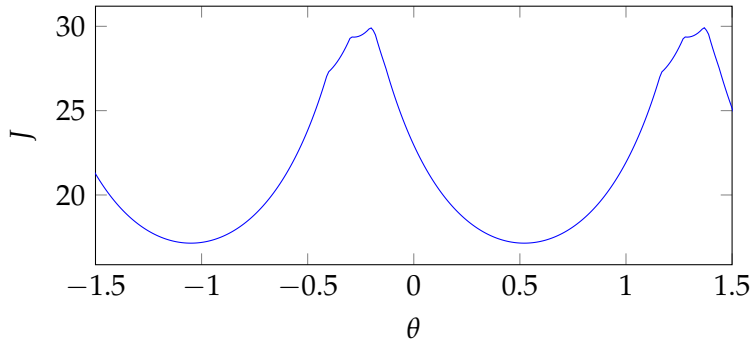

Figure 4.2: Cost depending on $\theta \in [-\pi/2, \pi/2]$ in the Givens rotation of §3.1 in the coordinate plane $(1,2)$.

Θ is composed of $K = \frac{N(N-1)}{2}$ θ with each θ representing a rotation of X in a coordinate plane. Thus with two eigenvectors $K = 1$, with three eigenvectors $K = 2$ and with four eigenvectors $K = 4$ thus 4 θ has to be found in order to minimize J.

Each θ is dependent of the other thus minimizing J gets increasingly costly depending on K. In the original code, we loop over Θ up to 200 times, we optimize the values Θ_k sequentially until J stabilizes itself and we then returns the cost and R.

This optimization is made using two methods: *numericalDerivative* and *trueDerivative*. In both cases we start with $\Theta = [0, 0, ..., 0]$ composed of K elements, the difference is how the optimization of Θ_k is made:

- In the numerical derivative, we compute the Givens rotation of X with Θ after updating Θ_k to $\Theta_k + \alpha$, we save the cost of the new rotation and then compute the rotation and its cost again but this time with Θ_k becoming $\Theta_k - \alpha$. If one of the new cost is better than the original cost, we save the Θ_k giving the best rotation. The value α is the step size, set to 0.1.
- In the true derivative, we update Θ_k by using the gradient descent scheme so that Θ_k becomes $\Theta_k - \alpha \nabla J|_{\Theta=\Theta_k}$ with $\alpha = 1e$.

These computations, made sequentially for every Θ_k up to 200 times until J stabilizes itself (i.e. $J_{i-2} - J < 0.0001 J_{i-2}$), has shown problems once implemented. The numerical derivative has an obvious limitation of having a fixed step size and the true derivative has not been able to cluster correctly all the datasets (§4.1) during my trials.

The solution was to create a new computation of the derivative: the true numerical derivative. The computation is the same as in the true derivative but, instead of computing ∇J following the appendix A of the original paper, we compute it numerically using a central difference [12]:

$$\nabla J = \frac{J|_{\Theta_{up}} - J|_{\Theta_{down}}}{2\alpha}$$

With Θ_{up} being Θ with Θ_k set to $\Theta_k + \alpha$, Θ_{down} being Θ with Θ_k set to $\Theta_k - \alpha$ and $\alpha = 0.001$.

k	J **(numerical derivative)**	J **(true derivative)**	J **(true numerical derivative)**
2	513.22889	513.18389	513.16030
3	513.85276	719.17141	512.00637
4	514.76266	653.59525	512.01187
5	600.01559	638.25511	528.90081
6	548.18117	692.18355	541.51188

Table 4.1: Cost J depending on k and the derivative used when computing STSC on the second dataset of the second row of the figure §4.1

The true numerical derivative combines the best of both algorithms: it is faster than the numerical derivative as it takes advantage of the gradient to find the best Θ and it gives better result than the true derivative. The results of the true derivative are on certain cases way worse than the numerical derivative for a reason that has not been found.

These problems could explain why the original implementation contains two implementations, the one in the paper having not been correctly implemented. The true numerical derivative is the one used in the evaluation chapter when the derivative is not specified.

To read the Scala implementation of the three derivatives, a milestone has been created on

GitHub containing them before leaving only the true numerical derivative [3].

4.3 Comparing the rotations, cost v. quality

One of the difference noticed between the paper and the code was the computation of the cost in the paper and of the quality in the code to compare the rotations of the eigenvectors. Both techniques have been implemented in two different branches of the Git repository of STSC and then compared:

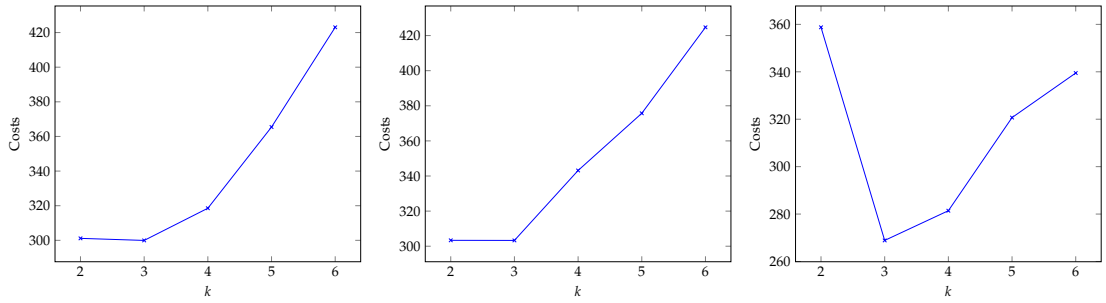

Figure 4.3: Costs for $k \in [2;6]$ clusters for the three first given datasets (20) using the numerical derivative.

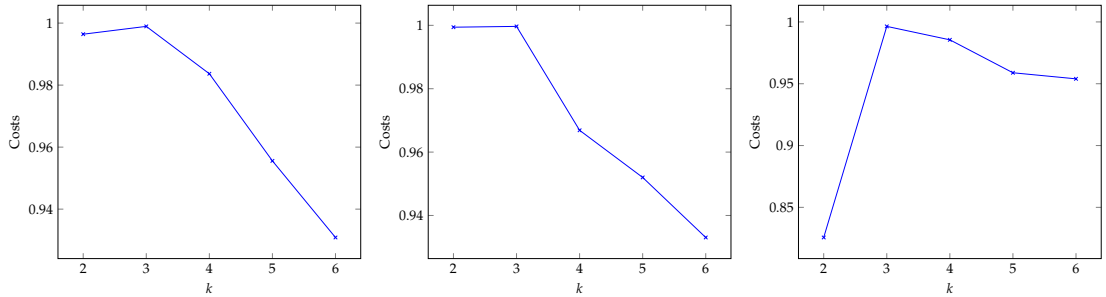

Figure 4.4: Qualities for $k \in [2;6]$ clusters for the same datasets, using the numerical derivative.

As we can see on the figures, costs and qualities are not on the same scale and opposite on the y axis. However, the difference between each quality or cost are similar if we compare them using percentages. The implementation developed for this thesis uses cost instead of quality to follow the paper.

In the original code, the comparison between qualities is made using the value 0.001. In the paper, this value is 0.01% of the previous cost. As the qualities go up to 1, it means that the code consider a difference up to $0.001 * 1 = 0.1\%x1$.

This percentage has been used when comparing the costs obtained through the numerical and true derivative but the new true numerical derivative has allowed to put back the original percentage of 0.01% when comparing the costs.

The evaluation of the differences is made in the next chapter, the final choices have been to:

- Use cost instead of quality, as in the paper.
- Consider costs up to 0.01% the same, as in the paper.

[3] `https://github.com/ArmandGrillet/stsc/releases/tag/v1.0-alpha`

- Use a new function called the *trueNumericalDerivative* to recover the aligning rotation. A mix of the two existing functions that offers better results as we have seen in §4.1.

These choices offer a computation (§6.1) that follows the original paper more precisely than the original code while offering the same results (§5.1). The algorithm is readable in the appendix A.

4.4 User interfaces created to test the algorithm

After evaluating the correctness of the algorithms on the given datasets (§5.1), a few user interfaces have been developed to test the algorithm.

4.4.1 stsc-1dcluster and stsc-2dcluster

The first thing to test was the clustering capabilities of the algorithm on a dataset containing two clusters sampled from an isotropic Gaussian (which is multivariate when working in two dimensions). The graphical user interfaces (GUIs) have been developed using ScalaFXML [4] in order to see the results and limits of the algorithm.

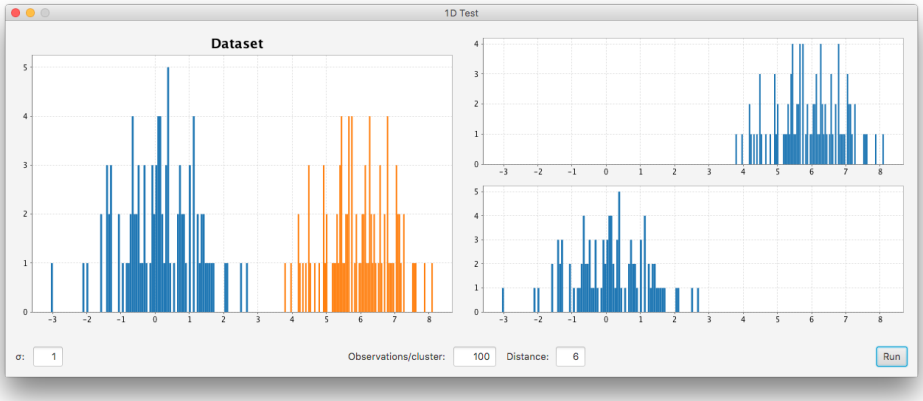

Figure 4.5: stsc-1dcluster, GUI made to see the results when clustering a dataset composed of two clusters sampled from an isotropic Gaussian. The inputs are the variance θ, the number of observations per cluster and the distance between them based on μ. The dataset on the left is the representation of the two clusters created with a different color to see which observation should on which cluster. The histograms on the right are the clusters found by STSC.

The code of every applications presented is available[5] on Github[6].

4.4.2 stsc-uicluster

A more general application has then developed to cluster datasets given as *.csv* files[7].

[4] https://github.com/vigoo/scalafxml
[5] https://github.com/ArmandGrillet/stsc-1dtestcluster
[6] https://github.com/ArmandGrillet/stsc-2dtestcluster
[7] https://github.com/ArmandGrillet/stsc-uicluster

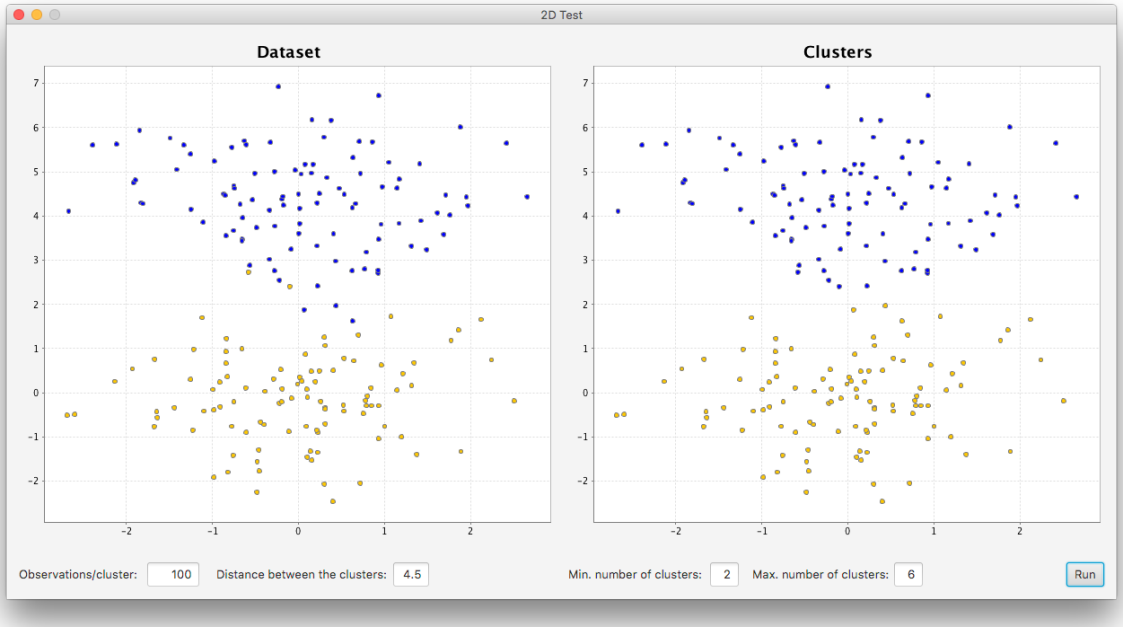

Figure 4.6: stsc-2dcluster, GUI made to see the results when clustering a dataset in two dimensions. The inputs are the number of observations per clusters, the distance between them and the minimum and maximum k used as inputs when clustering the dataset. The graph on the left is the representation of the two clusters created with a different color to see which observation is on which cluster. The graph on the right is the result of the clustering algorithm.

Figure 4.7: stsc-uicluster, GUI made to see the results when clustering a given dataset. The inputs are the minimum and maximum k used to cluster the given dataset. The dataset loaded is displayed on the left and the graph on the right represents the clusters created from it.

Unit tests have also been developed [8] but these interfaces have been implemented to get visual feedback.

4.5 Parallelizing the algorithm using Apache Spark

The main interest of the k-d tree (§3.2) is to get independent tiles and ensure that no cluster will be missed using the border width. Once we have independent tiles we can process them in parallel, and this is where we start using Apache Spark.

The core of Spark is its concept of Resilient Distributed Dataset (RDD) [18], this is the collection of elements that are processed in parallel. To use spark and RDDs a new function has been developed in the STSC library called $sparkCluster$.

This function takes a Spark context (i.e. the connection to the Spark cluster running on a machine or a cluster), the paths of a dataset and its k-d tree as *.csv* files, an output path for the result(s) and the minimum and maximum potential number of clusters per tile. We also let the user choose between having as an output only the centers of each cluster in the dataset or a cluster ID for each observation.

There are more inputs than in the sequential function that only takes a CSV or a matrix as an input with the minimum and maximum potential number of clusters in the entire cluster. The main reason behind this is that using Spark makes sense only when clustering big datasets divided using a k-d tree. This is also why we ask for dataset and the k-d tree to be given through paths instead of in-memory objects.

4.5.1 The parallelization: new objects, new concepts

Before parallelizing the algorithm, new classes have been created in the library representing the k-d tree, composed of a head Node and a border width. The Node object is the component of a tree, it contains a Tile, its $value$, and two nodes called $left$ and $right$. If $left$ and $right$ are null then the Node is a leaf.

A Tile is composed of $mins$ and $maxs$, values representing the edges of the tile in the dimensions of the dataset. These simple principles offer an implementation that is minimal but with useful functions such as $KDTree.owningTiles(Observation)$, operating with a time complexity of $O(logn)$, and conversions of the k-d tree into a *.csv* file and vice versa.

Before testing the Spark implementation, unit tests have been made to make sure that these objects work as expected.

4.5.2 Applying Spark on top of the sequential algorithm

The steps to run the algorithm in parallel, using the inputs described at the beginning of the section, are:

1. Creating a map of the smallest tiles in the k-d tree zipped with an index, giving us a unique id for each tile.

[8] `https://github.com/ArmandGrillet/stsc/tree/master/src/test/scala/stsc/sequential/unit/STSC`

2. Broadcasting the map, the *k*-d tree, the dimension of the tree and the minimum and maximum potential number of clusters per tile to each slave running in the Spark cluster.
3. Grouping the observations per tiles. This operations is done by reading the *.csv* file representing the dataset and mapping each observation to get a sequence of tuples (Tile, Observation). As an observation can be in more than one tile due to the border, we use a variant of map that flatten this result.
4. We then group the elements depending on the Tile to get a RDD where each element is all the observations in a tile (including the ones in its border).

Once we have this RDD, the sequential clustering algorithm is applied on each element (i.e. tile) in parallel. Two functions exist depending on what is the output requested by the user but they share the same core: computing the clusters of the element and checking if its center is in the tile or not.

If it is not the case, the observations of the cluster are removed from the RDD element. This operation ensures that one observation will not be in two clusters. Once we have all the clusters, we reduce the RDD into one directory using the Spark library and then merge each file returned by each cluster into one *.csv* or *.json* file using the Hadoop library and its merging capabilities.

4.5.3 Deploying the code in the cloud

To compare the sequential and parallel implementation, the algorithms have been run on a Google Compute Engine[9] instance having four CPUs and 10 GB of RAM.

Other solutions were available, such as creating a Spark cluster with real instances as workers. As Spark also offer a standalone mode[10] where one machine can be considered as a cluster, this is the solution used.

[9] `https://cloud.google.com/compute/`
[10] `http://spark.apache.org/docs/latest/spark-standalone.html`

5 Evaluation

5.1 Evaluating the sequential implementation

5.1.1 Testing the algorithm on the original datasets

To evaluate the sequential implementation, it has been tested on the datasets given with the original code (§4.1):

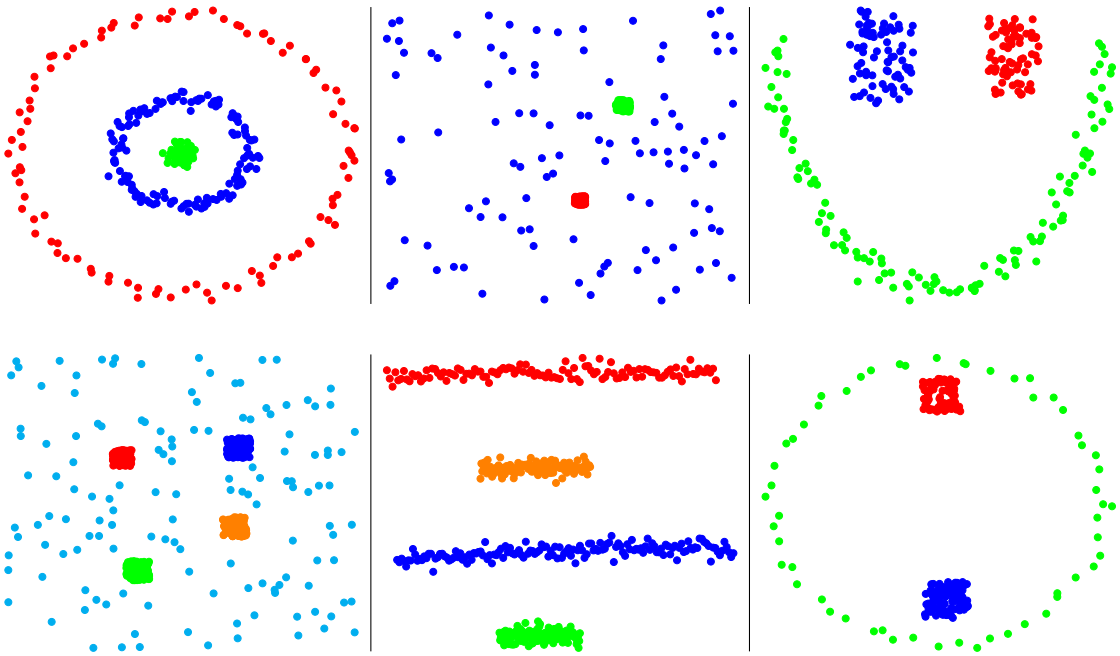

Figure 5.1: Clusters found using the new implementation of STSC, with $k \in [2, 6]$

As in the paper, the clustering result for these six datasets is correct. Clustering each dataset takes less than 1 second and the results do not variate when the algorithm is run dozens of times.

This evaluation proves that the new algorithm is not worse than the implementation made for the original paper. This evaluation can be tested by running the test *STSCTest.scala* available in the GitHub project stsc [1].

[1] `https://github.com/ArmandGrillet/stsc/`

5.1.2 Testing the limits of the algorithm in 1 and 2 dimensions

The implementations stsc-1dcluster and stsc-2d (§4.4.1) cluster as well as unit tests have been used to evaluate the quality of the clustering in a strict manner. This evaluation consisted in seeing the minimal distance between two clusters following a (multivariate) Gaussian distribution in order to be correctly clustered with $k \in [2, 6]$.

As the distributions change between two samplings, the clustering was considered correct if 5 datasets containing two clusters with the same distance (i.e. the distance between the μ of the two clusters) were correctly clustered 4 times.

In one dimension, the minimum distance between the two clusters is 5.5 while the minimum distance between two clusters in two dimensions is 5.8. Testing these distances with the GUIs previously implemented confirmed these results.

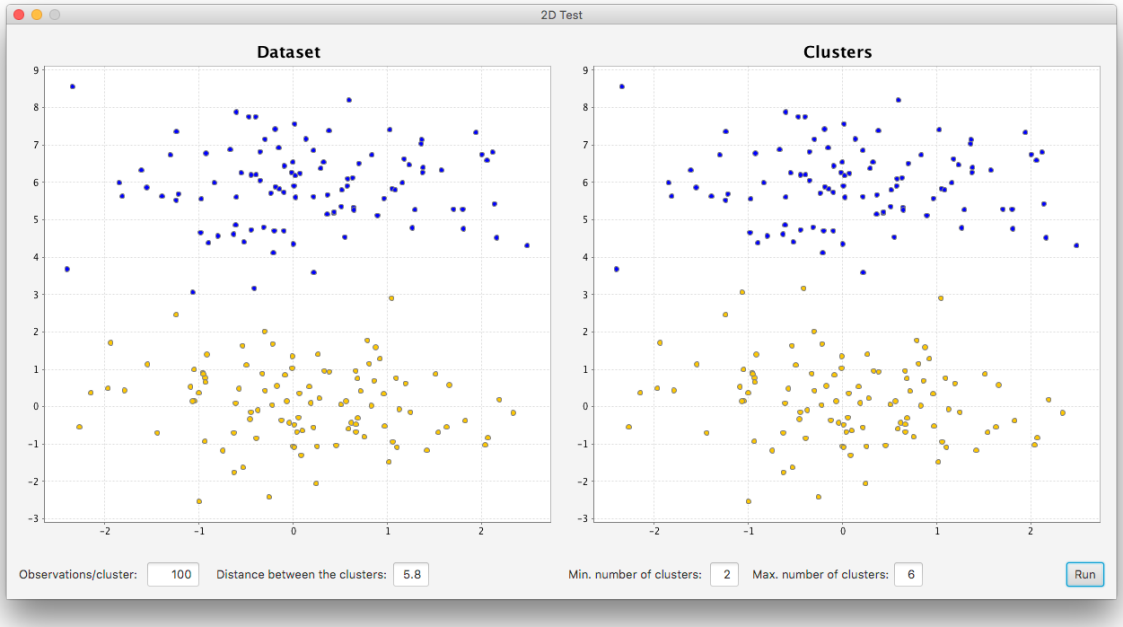

Figure 5.2: The limit in two dimensions displayed using stsc-2dcluster. We can see that two observations are not correctly clustered (they are yellow but they should be blue).

The main question once we have this information is how the algorithm handles datasets containing clusters that are even closer than the limits described above. The result was quite disappointing in one dimension: the algorithm tends to find too many clusters (between 4 to 6 during the evaluation).

In two dimensions, the behavior is different and the algorithm still finds two clusters albeit the observations are not correctly clustered. The algorithm appears in that case to work like related solutions, but there is one big problem.

As described when explaining the concepts of STSC (§3), the algorithm is not able to detect only one cluster in a dataset. Thus, if the dataset is only composed of two extremely close clusters following a multivariate Gaussian distribution, two clusters will be found whereas

other algorithms would correctly find only one cluster.

This problem is only important if there are only two clusters in the dataset as the algorithm works with a minimum value $k = 2$.

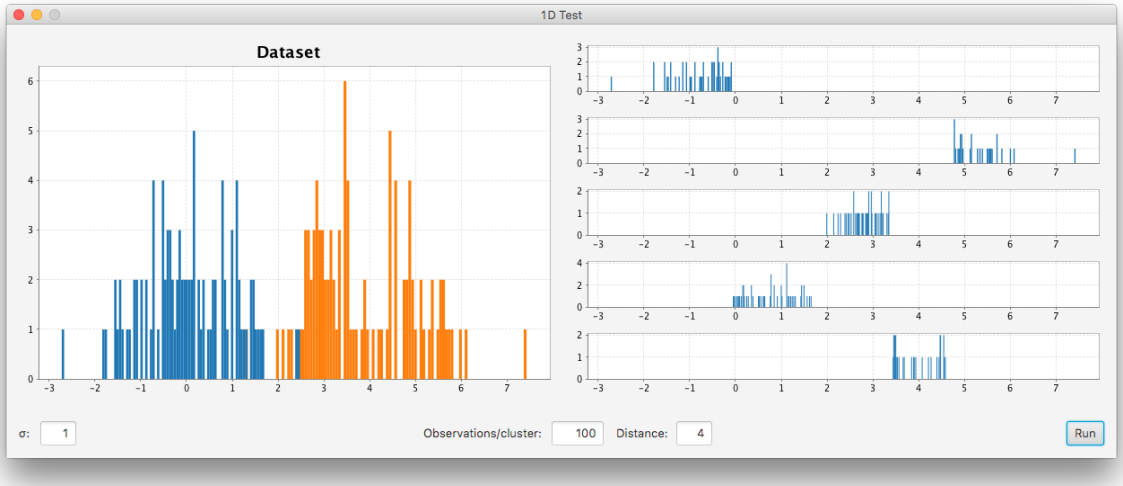

Figure 5.3: For two close clusters in one dimension, STSC considers 4 to be the best possible number of clusters.

5.1.3 Testing the importance of the minimum possible number of clusters

One subject that is not discussed in the original paper is the computation time depending on the minimum and maximum number of clusters in the dataset. For instance, is computing STSC for (minClusters, maxClusters) = (10, 20) faster than for (minClusters, maxClusters) = (2, 20)? The paper states that "We start by aligning the top two eigenvectors (as well as possible)" but the original code does not imply that we need to start with minClusters = 2.

To test the importance of this parameter, we use a dataset that contains 9 well separated clusters containing observations following mutlivariate Gaussian distributions. I have then look at the computation time when clustering this dataset with (minClusters, maxClusters) = (2, 10) and (minClusters, maxClusters) = (6, 10).

(minClusters, maxClusters)	**Average computation time of STSC**
(2, 10)	2,064s
(3, 10)	2,051s
(4, 10)	1,949s
(5, 10)	1,953s
(6, 10)	2,045s

Table 5.1: Average computation time of the sequential algorithm depending on $minClusters$ and $maxClusters$.

The costs K with $K \in [6, 10]$ are nearly identical in all cases, less than a 0.01% difference

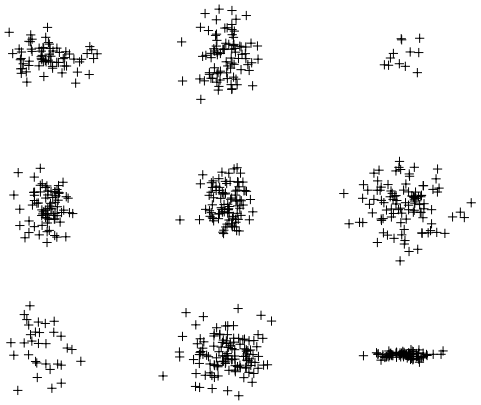

Figure 5.4: The dataset clustered for the evaluation of the importance of $minClusters$. This dataset has been created with a program developed for this thesis and used in the evaluation of the parallel implementation.

for the same K. The computation time goes down at the beginning but then go up, proving that the fact of not computing certain rotations by starting with more than 2 clusters does not compensate the cost of the first Givens rotation with many eigenvectors.

These results are in accordance with the original paper [31]: "The overall runtime is just slightly longer than aligning all the eigenvectors in a non-incremental way".

5.1.4 Testing the time complexity of the algorithm

The original paper does not mention the time complexity of the algorithm. Due to the use case behind my master thesis, it was something important to evaluate. In order to do the evaluation, I have created a simple Scala program creating a given number of clusters following a multivariate Gaussian distribution (giving results like §5.4).

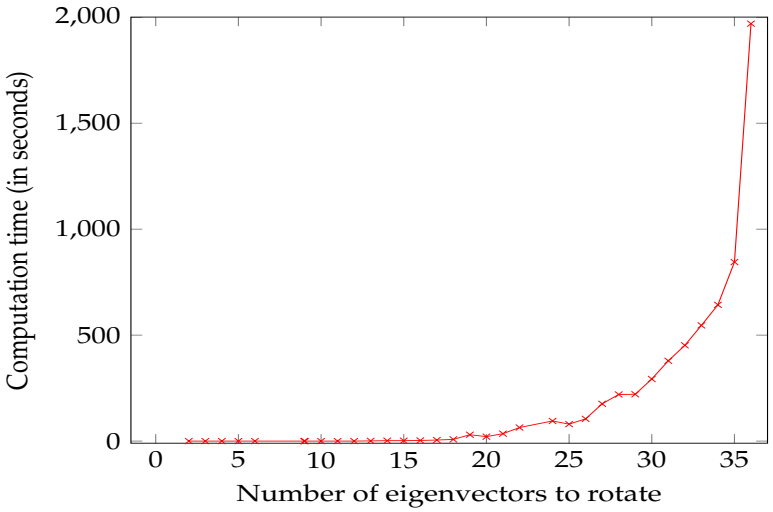

Figure 5.5: Time complexity of computing the best rotations of a number of eigenvectors from a dataset composed of 1000 observations divided in 50 clusters, starting with two eigenvectors.

From 20 eigenvectors, the time complexity starts to be of $O(x^2)$. These values show that the k-d tree will be essential for clustering datasets composed of many clusters.

5.2 Evaluating the parallel implementation

5.2.1 Comparing the sequential and Spark implementations

To do the evaluation, I have used a dataset in 2 dimensions composed of 36 clusters, with 100 observations in each. The sequential computation has been done with (minClusters, maxClusters) = (30, 40), the parallel computation has for parameters (minTileClusters, maxTileClusters) = (2, 12). The k-d tree used to test the Spark algorithm is composed of a border-width of 6 (the average width/height of a cluster being 10) and 8 leafs.

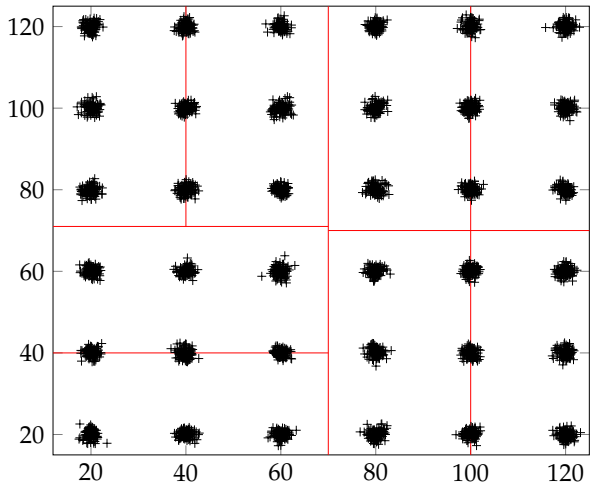

Figure 5.6: Dataset used to evaluate the parallel implementation and compare it with the sequential one. k-d tree represented by the red lines, computed and used to evaluate the Spark implementations.

The two evaluations of the Spark implementation using the dataset and the k-d tree has been made with:

1. 2 Spark workers having 2 cores and 4GB of RAM each.
2. 4 Spark workers having 1 core and 2GB of RAM each.

The goal was to compare the computation time and results of the sequential implementation. The results are:

Function used	Computation time	Best number of clusters found
Sequential clustering	18237s	36
Creation of the k-d tree	0.1838s	(Not applicable)
Parallel clustering (2 workers)	42.2309s	36
Parallel clustering (4 workers)	45.6635s	36

Table 5.2: Comparison of sequential and parallel clustering for a dataset of 3600 observations containing 36 clusters.

The comparison shows that the parallel implementation works and that its computation time

is way faster than the sequential algorithm. As we have seen in the previous evaluation (§5.1.4), computing the cost of a high number of cluster in a dataset takes a very long time thus dividing the biggest possible number of clusters in a dataset using a *k*-d tree is extremely efficient.

The fact that doing the clustering in parallel with 4 workers is slower than with 2 workers on a machine with 4 cores is strange. The only explanation can come from the fact that each worker has 2 cores when running the algorithm on Spark with 2 workers but, as the *k*-d tree is composed of 8 leafs, it is not logical. The tests of the parallel implementation have been run multiple times to ensure the results, the sequential implementation has only been run once due to its duration (more than 5 hours).

5.2.2 Limits of the *k*-d tree

There is one big problem with the *k*-d tree: it cannot be parallelized and the entire dataset has to be in memory, at least to do the first cut. This limitation means that a dataset that is too big has to be clustered by hand, or using other techniques such as using only one dimension to do the cut.

As we have seen in the previous section, cutting and clustering in parallel makes the computation faster by an order of magnitude. Now that the question of how to cluster a dataset with 36 groups is done, let us see how long takes the computation of a *k*-d tree for bigger datasets.

The evaluation consists in clustering three datasets containing 100, 1000 and 10000 clusters with 100 observations in each. The comparison will be done between the computation time of the *k*-d tree when creating it with leafs containing a maximum of 1000 observations (one of the two possible methods using the STSC library).

Number of clusters in the dataset	Computation time	Number of leafs
100	0,1106s	16
1000	1,1638s	128
10000	13,0635s	1024

Table 5.3: Computation time of a *k*-d tree depending on the size of the dataset. Dataset with 100 observations in each cluster, function run with $maxTileObservations = 1000$.

The time complexity for creating the *k*-d tree appears linear. The last thing to evaluate was the clustering using the *k*-d tree on a dataset containing 1000 clusters of 100 observations to see if it was feasible. The evaluation has been done on a Spark cluster with 2 workers (2 cores and 4GB of RAM each). The *k*-d tree has been set to a border width of 5 as the clusters have a length of around 5 in each dimension.

Running the clustering algorithm in that configuration took 8498 seconds, 2 times less than running the sequential algorithm on a 28 times smaller dataset (§5.2).

6 Conclusion

This thesis has been the opportunity to take a clustering algorithm created more than a decade ago and improve it for new use cases using new technologies.

The most important production made for this thesis is a new version of the self-tuning spectral clustering algorithm. This version is more readable, usable as a library and, more importantly, the first implementation that strictly follows the description made in the original paper Self-Tuning Spectral Clustering [31]. The sequential algorithm is readable in the appendix (§6.1).

This implementation could not have been done without understanding what is happening in each of the eight steps of the algorithm. The concepts are explained extensively in this thesis (§3), it give a new description of the algorithm and have permitted enhancements, e.g. concerning the incremental descent scheme used to find the best rotation of the eigenvectors.

The evaluations made for this thesis show that the algorithm is only suitable when clustering datasets containing a small number of clusters as the computation time becomes too important when clustering more than 20 clusters.

This is why parallelizing the algorithm makes sense. By combining the algorithm with a way to cut the dataset into tiles containing the same number of observations, we are able to cluster large datasets containing multiple small clusters.

Cutting a dataset adds new challenges such as how to handle clusters containing observations in two clusters. By using solutions like borders and implementing them, the algorithm is able to cluster large datasets containing more than 20 clusters in a few seconds.

This thesis has been focused on the use-case described in the introduction: how to cluster a map of the road signs in Europe? This kind of dataset, containing observations forming small clusters following a multivariate Gaussian distributions, can be clustered using the implementation. For datasets containing large clusters, the algorithm is not suitable.

Using Apache Spark to parallelize the clustering has been an efficient solution as the implementation uses the same core algorithm as the sequential function. By using libraries such as Spark, Hadoop and Breeze, the Scala code is focused on the algorithm and not implementations of standard elements such as matrices.

By giving a new implementation of the self-tuning spectral clustering algorithm, parallelizing it and testing its limits, use cases for the algorithm emerge. Compared to other clustering algorithms, it offers advantages when clustering small datasets. Spectral clustering allows clusters of any shapes and this algorithm allows the user to give as inputs only the dataset and a maximimum possible number of clusters.

Using the algorithm for big datasets seems less interesting, the computation of the biggest eigenvectors of the affinity matrix using standard value decomposition takes a large amount of heap space and rotating them (§3.1.5) takes too much time. The solution offered by cutting the dataset limits the algorithm to a few use cases such as the one described in the introduction.

6.1 Future work

The work behind this thesis has been done in two steps: first the sequential algorithm then the Spark implementation. Spark 2.0 has been released just before starting the work on the Spark implementation and it has a library to do linear algebra with the same features as Breeze. Due to this update, it would be interesting to remove the Breeze dependency to only be dependent of the Spark library.

Another change that should be applied to the library developed concerns the *k*-d tree design. The main element to change is the border width used to handle clusters that are in two tiles. Finding a solution to compute the border width dynamically depending on the dataset analyzed would be interesting. Another update would be to have a border width in each dimension of the dataset.

The evaluation of the algorithm has been made on relatively small generated datasets due to the need to compare the sequential and parallel implementations. In the future, it would be interesting to test the parallel implementation on datasets resulting of a concrete use case instead of a computed datasets made of well spaced multivariate Gaussian distributions.

Concerning tests, they have mainly been done on datasets of one or two dimensions The different graphical user interfaces also uses the library only in one and two dimensions in order to display the results to the user. The original paper only mentions the dimensions of the dataset when computing the local scaling (§3.1.1) and no research has been done on using the algorithm for datasets of higher dimensionality.

List of Tables

List of Figures

Bibliography

[1] Osama Abu Abbas. "Comparisons Between Data Clustering Algorithms." In: *Int. Arab J. Inf. Technol.* 5.3 (2008), pp. 320–325. URL: http://dblp.uni-trier.de/db/journals/iajit/iajit5.html#Abbas08.

[2] M. Ankerst et al. "OPTICS: Ordering Points To Identify the Clustering Structure". In: *Proc. ACM SIGMOD Int. Conf. on Management of Data (SIGMOD'99)*. Philadelphia, PA, 1999, pp. 49–60.

[3] Michael Armbrust et al. "Spark sql: Relational data processing in spark". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM. 2015, pp. 1383–1394.

[4] Shai Ben-David, Ulrike Von Luxburg, and Dávid Pál. "A sober look at clustering stability". In: *International Conference on Computational Learning Theory*. Springer. 2006, pp. 5–19.

[5] Jon Louis Bentley. "Multidimensional Divide-and-Conquer." In: *Commun. ACM* 23.4 (1980), pp. 214–229. URL: http://dblp.uni-trier.de/db/journals/cacm/cacm23.html#Bentley80.

[6] Sébastien Bubeck, Marina Meilă, and Ulrike von Luxburg. "How the initialization affects the stability of the k-means algorithm". In: *ESAIM: Probability and Statistics* 16 (2012), pp. 436–452.

[7] F. R. K. Chung. *Spectral Graph Theory*. American Mathematical Society, 1997.

[8] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters". In: *Communications of the ACM* 51.1 (2008), pp. 107–113.

[9] Martin Ester et al. "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise". In: *Proc. of 2nd International Conference on Knowledge Discovery and Data Mining*. 1996, pp. 226–231.

[10] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. "The Google File System". In: *SIGOPS Oper. Syst. Rev.* 37.5 (Oct. 2003), pp. 29–43.

[11] Vivek K Goyal and Martin Vetterli. "Block transform adaptation by stochastic gradient descent". In: *In IEEE Dig. Sig. Proc. Workshop*. 1998, p. 75.

[12] Francis Martin Henderson. *Open channel flow*. Vol. Macmillan series in civil engineering. Macmillan, 1996.

[13] Anil K Jain, M Narasimha Murty, and Patrick J Flynn. "Data clustering: a review". In: *ACM computing surveys (CSUR)* 31.3 (1999), pp. 264–323.

[14] Virginia Klema and Alan Laub. "The singular value decomposition: Its computation and some applications". In: *IEEE Transactions on automatic control* 25.2 (1980), pp. 164–176.

[15] Stuart Lloyd. "Least squares quantization in PCM". In: *IEEE transactions on information theory* 28.2 (1982), pp. 129–137.

[16] Ulrike von Luxburg. "A Tutorial on Spectral Clustering". In: *CoRR* abs/0711.0189 (2007). URL: http://arxiv.org/abs/0711.0189.

[17] J. MacQueen. "Some methods for classification and analysis of multivariate observations". In: *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*. Berkeley, Calif.: University of California Press, 1967, pp. 281–288. URL: http://projecteuclid.org/euclid.bsmsp/1200512992.

[18] Michael J. Franklin Matei Zaharia Mosharaf Chowdhury. "Spark: Cluster Computing with Working Sets". In: (2009). URL: https://amplab.cs.berkeley.edu/wp-content/uploads/2011/06/Spark-Cluster-Computing-with-Working-Sets.pdf.

[19] Xiangrui Meng et al. "Mllib: Machine learning in apache spark". In: (2016).

[20] B. Mohar. "The Laplacian spectrum of graphs". In: *Graph Theory, Combinatorics, and Applications* 2 (1991), pp. 871–898.

[21] Todd K Moon. "The expectation-maximization algorithm". In: *IEEE Signal processing magazine* 13.6 (1996), pp. 47–60.

[22] Andrew W Moore. "An intoductory tutorial on kd-trees". In: (1991). URL: http://www.autonlab.org/autonweb/14665/version/2/part/5/data/moore-tutorial.pdf.

[23] Andrew Y. Ng, Michael I. Jordan, and Yair Weiss. "On Spectral Clustering: Analysis and an algorithm". In: (2001), pp. 849–856.

[24] *Self-Tuning Spectral Clustering - Demos*. Accessed: 2016-09-02. URL: http://www.vision.caltech.edu/lihi/Demos/SelfTuningClustering.html.

[25] Jianbo Shi and Jitendra Malik. "Normalized cuts and image segmentation". In: *IEEE Transactions on pattern analysis and machine intelligence* 22.8 (2000), pp. 888–905.

[26] Konstantin Shvachko et al. "The Hadoop Distributed File System". In: *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*. IEEE. 2010, pp. 1–10.

[27] Gilbert Strang. *Introduction to Linear Algebra*. Fourth. Wellesley, MA: Wellesley-Cambridge Press, 2009.

[28] Vinod Kumar Vavilapalli et al. "Apache hadoop YARN: Yet Another Resource Negotiator". In: *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM. 2013, p. 5.

[29] Reynold S Xin et al. "GraphX: Unifying data-parallel and graph-parallel analytics". In: *arXiv preprint arXiv:1402.2394* (2014).

[30] Maya R. Gupta Yihua Chen. "EM demystified: An expectation-maximization tutorial". In: *Electrical Engineering* (2010).

[31] Lihi Zelnik-Manor and Pietro Perona. "Self-Tuning Spectral Clustering". In: (2004), pp. 1601–1608. URL: http://dblp.uni-trier.de/db/conf/nips/nips2004.html#Zelnik-ManorP04.

Appendices

The sequential algorithm

```
1  package stsc
2
3  import breeze.linalg.{DenseMatrix, DenseVector, argmax, max, sum, svd, *}
4  import breeze.linalg.functions.euclideanDistance
5  import breeze.numerics.{abs, cos, pow, sin, sqrt}
6  import breeze.optimize._
7
8  import scala.collection.immutable.SortedMap
9  import scala.math.exp
10 import scala.util.control.Breaks.{break, breakable}
11
12 object STSC {
13     // Cluster a given dataset using a self-tuning spectral clustering algorithm
14     def cluster(dataset: DenseMatrix[Double], minClusters: Int = 2, maxClusters:
           Int = 6): (Int, Map[Int, Double], Array[Int]) = {
15         // Three possible exceptions: empty dataset, minClusters less than 0,
               minClusters more than maxClusters.
16         if (dataset.rows == 0) {
17             throw new IllegalArgumentException("The dataset does not contains
                   any observations.")
18         }
19         if (minClusters < 0) {
20             throw new IllegalArgumentException("The minimum number of clusters
                   has to be positive.")
21         }
22         if (minClusters > maxClusters) {
23             throw new IllegalArgumentException("The minimum number of clusters
                   has to be inferior to the maximum number of clusters.")
24         }
25
26         // Compute local scale (step 1).
27         val distances = euclideanDistances(dataset)
28         val scale = localScale(distances, 7) // In the original paper we use the
               7th neighbor to create a local scale.
29
30         // Build locally scaled affinity matrix (step 2).
31         val scaledMatrix = locallyScaledAffinityMatrix(distances, scale)
32
33         // Build the normalized affinity matrix (step 3)
34         val normalizedMatrix = normalizedAffinityMatrix(scaledMatrix)
```

```
35
36          // Compute the largest eigenvectors (step 4)
37          val largestEigenvectors = svd(normalizedMatrix).leftVectors(::, 0 until
                maxClusters)
38
39          var cBest = minClusters // The best group number.
40          var currentEigenvectors = largestEigenvectors(::, 0 until minClusters)
                // We only take the eigenvectors needed for the number of clusters.
41          var (cost, rotatedEigenvectors) = bestRotation(currentEigenvectors)
42          var costs = Map(minClusters -> cost) // List of the costs.
43          var bestRotatedEigenvectors = rotatedEigenvectors // The matrix of
                rotated eigenvectors having the minimal cost.
44
45          for (k <- minClusters until maxClusters) { // We get the cost of stsc
                for each possible number of clusters.
46              val eigenvectorToAdd = largestEigenvectors(::, k).toDenseMatrix.t //
                     One new eigenvector at each turn.
47              currentEigenvectors = DenseMatrix.horzcat(rotatedEigenvectors,
                    eigenvectorToAdd) // We add it to the already rotated
                    eigenvectors.
48              val (tempCost, tempRotatedEigenvectors) = bestRotation(
                    currentEigenvectors)
49              costs += (k + 1 -> tempCost) // Add the cost to the map.
50              rotatedEigenvectors = tempRotatedEigenvectors // We keep the new
                    rotation of the eigenvectors.
51
52              if (tempCost <= cost * 1.0001) {
53                  bestRotatedEigenvectors = rotatedEigenvectors
54                  cBest = k + 1
55              }
56              if (tempCost < cost) {
57                  cost = tempCost
58              }
59          }
60
61          val orderedCosts = SortedMap(costs.toSeq:_*) // Order the costs.
62          val absoluteRotatedEigenvectors = abs(bestRotatedEigenvectors)
63          val z = argmax(absoluteRotatedEigenvectors(*, ::)).toArray // The
                alignment result (step 8), conversion to array due to https://issues.
                scala-lang.org/browse/SI-9578
64          return (cBest, orderedCosts, z)
65      }
66
67      // Returns the euclidean distances of a given dense matrix.
68      private[stsc] def euclideanDistances(matrix: DenseMatrix[Double]):
            DenseMatrix[Double] = {
69          val distanceMatrix = DenseMatrix.zeros[Double](matrix.rows, matrix.rows)
                // Distance matrix, size rows x rows.
70
71          for (i <- 0 until matrix.rows) {
72              for (j <- i + 1 until matrix.rows) {
73                  distanceMatrix(i, j) = euclideanDistance(matrix(i, ::).t, matrix
```

```
                    (j, ::).t) // breeze.linalg.functions.euclideanDistance
74                  distanceMatrix(j, i) = distanceMatrix(i, j) // Symmetric matrix.
75              }
76          }
77
78          return distanceMatrix
79      }
80
81      // Returns the local scale as defined in the original paper, a vector
            containing the Kth nearest neighbor for each observation.
82      private[stsc] def localScale(distanceMatrix: DenseMatrix[Double], k: Int):
            DenseVector[Double] = {
83          if (k > distanceMatrix.cols - 1) {
84              throw new IllegalArgumentException("Not enough observations (" +
                    distanceMatrix.cols + ") for k (" + k + ").")
85          } else {
86              val localScale = DenseVector.zeros[Double](distanceMatrix.cols)
87
88              for (i <- 0 until distanceMatrix.cols) {
89                  val sortedDistances = distanceMatrix(::, i).toArray.sorted //
                        Ordered distances.
90                  localScale(i) = sortedDistances(k) // Kth nearest distance, the
                        0th neighbor is always 0 and sortedVector(1) is the first
                        neighbor
91              }
92
93              return localScale
94          }
95      }
96
97      // Returns a locally scaled affinity matrix using a distance matrix and a
            local scale
98      private[stsc] def locallyScaledAffinityMatrix(distanceMatrix: DenseMatrix[
            Double], localScale: DenseVector[Double]): DenseMatrix[Double] = {
99          val affinityMatrix = DenseMatrix.zeros[Double](distanceMatrix.rows,
                distanceMatrix.cols) // Distance matrix, size rows x cols.
100
101         for (i <- 0 until distanceMatrix.rows) {
102             for (j <- i + 1 until distanceMatrix.rows) {
103                 affinityMatrix(i, j) = -pow(distanceMatrix(i, j), 2)
104                 affinityMatrix(i, j) /= (localScale(i) * localScale(j))
105                 affinityMatrix(i, j) = exp(affinityMatrix(i, j))
106                 affinityMatrix(j, i) = affinityMatrix(i, j)
107             }
108         }
109
110         return affinityMatrix
111     }
112
113     // Returns the euclidean distance of a given dense matrix.
114     private[stsc] def normalizedAffinityMatrix(scaledMatrix: DenseMatrix[Double
            ]): DenseMatrix[Double] = {
```

```
115         val diagonalVector = DenseVector.tabulate(scaledMatrix.rows){i => 1 /
                sqrt(sum(scaledMatrix(i, ::))) } // Sum of each row, then power -0.5.
116         val normalizedMatrix = DenseMatrix.zeros[Double](scaledMatrix.rows,
                scaledMatrix.cols)
117
118         for (i <- 0 until normalizedMatrix.rows) {
119             for (j <- i + 1 until normalizedMatrix.cols) {
120                 normalizedMatrix(i, j) = diagonalVector(i) * scaledMatrix(i, j)
                        * diagonalVector(j)
121                 normalizedMatrix(j, i) = normalizedMatrix(i, j)
122             }
123         }
124
125         return normalizedMatrix
126     }
127
128     // Step 5 of the self-tuning spectral clustering algorithm, recover the
            rotation R which best aligns the eigenvectors.
129     private[stsc] def bestRotation(eigenvectors: DenseMatrix[Double]): (Double,
            DenseMatrix[Double]) = {
130         var nablaJ, cost = 0.0 // Variables used to recover the aligning
                rotation.
131         var newCost, old1Cost, old2Cost = 0.0 // Variables to compute the
                descend through true derivative.
132
133         val bigK = eigenvectors.cols * (eigenvectors.cols - 1) / 2
134         var theta, thetaNew = DenseVector.zeros[Double](bigK)
135
136         cost = j(eigenvectors)
137         old1Cost = cost
138         old2Cost = cost
139
140         breakable {
141             for (i <- 0 until 200) { // Max iterations = 200, as in the original
                    paper code.
142                 for (k <- 0 until theta.length) { // kth entry in the list
                        composed of the (i, j) indexes
143                     val alpha = 0.001
144                     nablaJ = numericalQualityGradient(eigenvectors, theta, k,
                            alpha)
145                     thetaNew(k) = theta(k) - alpha * nablaJ
146                     newCost = j(givensRotation(eigenvectors, thetaNew))
147
148                     if (newCost < cost) {
149                         theta(k) = thetaNew(k)
150                         cost = newCost
151                     } else {
152                         thetaNew(k) = theta(k)
153                     }
154                 }
155
156                 // If the new cost is not that better, we end the rotation.
```

```
157                 if (i > 2 && (old2Cost - cost) < (0.0001 * old2Cost)) {
158                     break
159                 }
160                 old2Cost = old1Cost
161                 old1Cost = cost
162             }
163         }
164 
165         val rotatedEigenvectors = givensRotation(eigenvectors, thetaNew) // The
               rotation using the "best" theta we found.
166         return (cost, rotatedEigenvectors)
167     }
168 
169     // Return the cost of a given rotation, follow the computation in the
           original paper code.
170     private[stsc] def j(matrix: DenseMatrix[Double]): Double = {
171         val squareMatrix = matrix :* matrix
172         return sum(sum(squareMatrix(*, ::)) / max(squareMatrix(*, ::))) // Sum
               of the sum of each row divided by the max of each row.
173     }
174 
175     // The numerical quality gradient given a matrix, a theta, the theta_k we
           want to update and the step size h.
176     private[stsc] def numericalQualityGradient(matrix: DenseMatrix[Double],
           theta: DenseVector[Double], k: Int, h: Double): Double = {
177         theta(k) = theta(k) + h
178         val upRotation = givensRotation(matrix, theta)
179         theta(k) = theta(k) - 2 * h // -2 * because we cancel the previous
               operation.
180         val downRotation = givensRotation(matrix, theta)
181         return (j(upRotation) - j(downRotation)) / ( 2 * h)
182     }
183 
184     // Givens rotation of a matrix depending on theta.
185     private[stsc] def givensRotation(matrix: DenseMatrix[Double], theta:
           DenseVector[Double]): DenseMatrix[Double] = {
186         // Find the coordinate planes (i, j).
187         var i, j = 0
188         val ij = List.tabulate(theta.length)(_ => {
189             j += 1
190             if (j >= matrix.cols) {
191                 i += 1
192                 j = i + 1
193             }
194             (i, j)
195         })
196 
197         val g = DenseMatrix.eye[Double](matrix.cols) // Create an empty identity
               matrix.
198 
199         var tt, uIk = 0.0
200         for (k <- 0 until theta.length) {
```

```
201             tt = theta(k)
202             for (i <- 0 until matrix.cols) {
203                 uIk = g(i, ij(k)._1) * cos(tt) - g(i, ij(k)._2) * sin(tt)
204                 g(i, ij(k)._2) = g(i, ij(k)._1) * sin(tt) + g(i, ij(k)._2) * cos
                        (tt)
205                 g(i, ij(k)._1) = uIk
206             }
207         }
208         return matrix * g
209     }
210 }
```