Universidad de Alcalá
Escuela Politécnica Superior

Grado en Ingeniería Informática

Bachelor's Thesis

Development of a R package to facilitate the learning of clustering techniques

Author: Eduardo Ruiz Sabajanes

Advisor: Juan José Cuadrado Gallego

2023

UNIVERSIDAD DE ALCALÁ

ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática

Bachelor's Thesis

Development of a R package to facilitate the learning of clustering techniques

Author: Eduardo Ruiz Sabajanes

Advisor: Juan José Cuadrado Gallego

Tribunal:

President: _____.

1st Vocal: _____.

2nd Vocal: _____.

Deposit date: September 25th, 2023

Abstract

This project explores the development of a tool, in the form of a R package, to ease the process of learning clustering techniques, how they work and what their pros and cons are. This tool should provide implementations for several different clustering techniques with explanations in order to allow the student to get familiar with the characteristics of each algorithm by testing them against several different datasets while deepening their understanding of them through the explanations. Additionally, these explanations should adapt to the input data, making the tool not only adept for self-regulated learning but for teaching too.

Keywords: R, Clustering, Machine Learning, Data Science, Teaching.

Contents

Abstract **v**

Contents **vii**

List of Figures **ix**

List of Algorithms **xi**

List of Acronyms **xvi**

1 Introduction **1**

1.1 Motivation . . . 1

1.1.1 Massive amounts of data . . . 1

1.1.2 Hands-on learning . . . 2

1.1.3 Self-regulated learning . . . 2

1.1.4 Problem: A-priori knowledge . . . 3

1.2 Project proposition . . . 3

1.3 Objectives . . . 3

1.3.1 General objectives . . . 3

1.3.2 Specific objectives . . . 4

1.4 Structure of the document . . . 4

2 State of the Art **5**

2.1 Partitional Clustering . . . 6

2.1.1 Hard/Crisp . . . 7

2.1.1.1 Square Error . . . 7

2.1.1.2 Model-based Clustering . . . 8

2.1.1.3 Graph-theoretic Clustering . . . 9

2.1.1.4 Density-based Clustering . . . 10

2.1.1.5 Subspace Clustering . . . 11

2.1.1.6 Search-based Clustering . . . 11

2.1.2 Mixture Resolving Algorithms . . . 14
2.1.2.1 Expectation Maximization . . . 14
2.2 Hierarchical Clustering . . . 15
2.2.1 Agglomerative Clustering . . . 16
2.2.2 Divisive Clustering . . . 17
2.2.3 Implementations that Improve on Hierarchical Clustering . . . 19

3 Implementation **21**
3.1 Developing R packages . . . 21
3.2 Algorithms . . . 22
3.2.1 K-Means . . . 22
3.2.2 DBSCAN . . . 26
3.2.3 Gaussian Mixture . . . 28
3.2.4 Agglomerative Hierarchical Clustering . . . 30
3.2.5 Divisive Hierarchical Clustering . . . 33
3.3 Automatic explanations . . . 35

4 Results **37**
4.1 Toy Datasets . . . 37
4.2 Algorithm Characteristics . . . 37
4.2.1 K-Means . . . 38
4.2.2 DBSCAN . . . 39
4.2.3 Gaussian Mixture . . . 39
4.2.4 Agglomerative Hierarchical Clustering . . . 40
4.2.5 Divisive Hierarchical Clustering . . . 41
4.3 Algorithm comparison . . . 43
4.4 Automatic explanations . . . 45
4.5 Uploading a package to CRAN . . . 47

5 Conclusions and Future Work **49**

Bibliography **51**

Appendix A Source Code Listings **61**
A.1 K-Means . . . 61
A.2 DBSCAN . . . 66
A.3 Gaussian Mixture Model with Expectation Maximization . . . 71
A.4 Agglomerative Hierarchical Clustering . . . 76
A.5 Divisive Hierarchical Clustering . . . 80
A.6 Auxiliary functions . . . 84

List of Figures

2.1 Taxonomy of clustering algorithms [27] 5
2.2 A partition with $n = 500$ and $k = 4$ 6
2.3 Density-based clusterization example 10
2.4 Mixture resolving clusterization example 15
2.5 Dendrogram representation for the hierarchical clustering of eight objects 16

3.1 Package development workflow with devtools 22

4.1 Toy Datasets 38
4.2 K-Means - Drawbacks 39
4.3 DBSCAN - Pros & Cons 40
4.4 GMM with EM - Characteristics 41
4.5 Agglomerative Hierarchical Clustering - Linkage strategies 42
4.6 Clusterization of every toy dataset with every algorithm 44
4.7 Automatic explanation for AHC (Part 1) 46
4.8 Automatic explanation for AHC (Part 2) 46
4.9 Automatic explanation for AHC (Part 3) 47
4.10 Automatic explanation for AHC (Part 4) 47

List of Algorithms

3.1 K-Means 23
3.2 K-Means - Random initialization 24
3.3 K-Means - K-Means++ initialization 24
3.4 Density-Based Spatial Clustering of Applications with Noise 26
3.5 DBSCAN - `expand` procedure 27
3.6 Gaussian Mixture Expectation Maximization 29
3.7 Agglomerative Hierarchical Clustering 32
3.8 Divisive Hierarchical Clustering 34

List of Source Codes

A.1 R implementation of the K-Means algorithm 66
A.2 R implementation of the DBSCAN algorithm 71
A.3 R implementation of the GMM with EM algorithm 76
A.4 R implementation of the AHC algorithm 80
A.5 R implementation of the DHC algorithm 84
A.6 Auxiliary functions to log the explanations 85

List of Acronyms

ABC Artificial Bee Colony Optimization Algorithm.

ACO Ant Colony Optimization.

AHC Agglomerative Hierarchical Clustering.

BEA Bacterial Evolutionary Algorithm.

BFS Breadth First Search.

BIC Bayesian Information Criterion.

BIRCH Balanced Iterative Reducing and Clustering Using Hierarchies.

CF Cluster Features.

CLICK Cluster Identification via Connectivity Kernels.

CRAN Comprehensive R Archive Network.

CURE Clustering Using Representative.

DBSCAN Density Based Spatial Clustering of Applications with Noise.

DCPSO Dynamic Clustering Particle Swarm Optimization.

DE Differential Evolution.

DENCLUE Density-based Clustering.

DHC Divisive Hierarchical Clustering.

DIVFRP Reference-point-based dissimilarity measure.

DTG Delaunay Triangulation Graph.

EM Expectation Maximization.

FA Firefly Algorithm.

FC Flow Cytometry.

FGKA Fast Genetic K-Means Algorithm.

GA Genetic Algorithm.

GGA Genetically Guided Algorithm.

GKA Genetic K-Means Algorithm.

GMM Gaussian Mixture Model.

GMM with EM Gaussian Mixture Model with Expectation Maximization.

HGCUDF Hierarchical grid clustering using data field.

IDE Integrated Development Environment.

IDPSO Improved particle optimizer.

IGKA Incremental Genetic K-means Algorithm.

IWO Invasive Weed Optimization.

K-NN K Nearest Neighbor.

MCA Multiple Correspondence Analysis.

MDA Maximum Dependency of Attributes.

MGR Mean Gaian Ratio.

MLE Maximum Likelihood Estimation.

MMR Min-Min-Roughness.

MNN Mutual Nearest Neighbors.

MNV Mutual Neighborhood Value.

MVP Minimum Viable Product.

OPTICS Ordering Points To Identify the Clustering Structure.

PSO Particle Swarm Optimization.

SI Swarm Intelligence.

SOS Symbiotic Organisms Search.

SWIFT Scalable Weighted Iterative Flow-clustering Technique.

TR Total Roughness.

TS Tabu Search.

TWCV Total Within Cluster Variation.

VNS Variable Neighborhood Search.

Chapter 1

Introduction

Cluster analysis or clustering is the task of assigning instances to classes that are not defined *a priori* and that are supposed to somehow reflect the underlying structure of the entities that the data represents i.e. grouping a set of objects in such a way that objects in the same group (called a cluster) are more similar to each other than to those in other groups (clusters) [1]. This subfield of data science, is used in many fields of study like pattern recognition, image analysis, information retrieval, bioinformatics, data compression, computer graphics and machine learning; and is now deeply rooted in our daily lives, from recommendation engines like those of online stores, to ocular disease detection systems [2]. In this chapter, we will discuss the motivation behind this work, the problem statement, the objectives and the structure of the document.

1.1 Motivation

1.1.1 Massive amounts of data

We live in the age of information. Everyday, millions of new data points like scientific articles, blog posts, discussions, social network comments, etc. are generated. Having the capability to go through that amount of data and classify it, would enable us to study the relations between the different data points and classes, effectively extracting knowledge from that data. This has turned clusterization into a thriving field of study, with many different algorithms being developed and improved every day.

In addition to this, there are many public datasets composed of synthetic or real-world data. Some of these datasets can be found in the UCI Machine Learning Repository [3] and among the most popular we can find the following: "Online Retail II", a real online retail transaction data set of two years [4] (1067371 points, 8 dimensions); "US Census Data (1990)", a data set containing a one percent sample of the Public Use Microdata Samples (PUMS) person records drawn from the full 1990 census sample [5] (2458285 points, 68 dimensions); or "Obesity", a dataset including data for the estimation of obesity levels in individuals from the countries of Mexico, Peru and Colombia, based on their eating habits and physical condition [6]. These are some of the datasets professionals of the data science sector use to test the performance of clustering algorithms, and are used as a benchmark to compare different algorithms. Additionally, since these datasets are open to the public, they can also be used by practitioners to improve their algorithm and hyper-parameter choice-making before applying them to real world problems.

All of this comes to prove that there is no problem when it comes to the amount of data. On the one hand there is a clear need for data analysts who can extract knowledge from the data, and on the other

hand there are many datasets available to test and improve the algorithms. This is why we believe that this is a good time to study the field of clustering and to develop new algorithms that can help us extract this knowledge.

1.1.2 Hands-on learning

In traditional classes, students may understand concepts incorrectly which would lead into misinterpreting them i.e. correct transfer of knowledge can not be guaranteed. One of the ways in which this problem can be solved is by using hands-on learning.

Hands-on is a way of teaching where students have resources for testing what they have learned i.e. students learn by doing [7]. It is also defined as learning with materials where students modify, handle and test the learning with materials [7]. It is based on the idea that the best way to learn something is through experience. The hands-on approach to teaching brings lab tasks to classrooms or vice-versa, where students get to use, change and test materials and observe the working modules of their learning [8]. In traditional teaching, the teacher is at the center of the learning process i.e. the teacher stands at the front of the room imparting and transferring their knowledge to students, while in hands-on learning, the students are the ones at the center i.e. students are actively involved trying out what they have learned, with the objective of creating interest in the students and promoting critical thinking [8].

On one hand, students develop a greater interest towards the course they are taking when the course implements hands-on teaching [9]. Interest can be defined as the students-subject relationship that is generated during the student's interaction with the subject [10], [11]. Student interest is an important factor which can determine learning quality and evaluation results [12]. Students also feel that hands-on sessions are more interesting than teacher instructions, watching videos or listening to audios [13].

On the other hand, the interest generated in hands-on can be varying, depending mainly on the student's involvement which is based on their positive cognition and emotions [10], [14]. This means that, as long as the students have a joyful experience, the interest towards the course will be higher. However, may they have a unsatisfactory experience (e.g. students are unable to complete the tasks), this will take a toll on the overall interest [11].

1.1.3 Self-regulated learning

Self-regulation is a system of conscious, personal management that involves the process of guiding one's own thoughts, behaviours and feelings to reach goals. It consists of several stages. In these stages, individuals must function as contributors to their own motivation, behaviour and development within a network of reciprocally interacting influences.

Self-regulated learning is one of the domains of self-regulation, and is aligned most closely with educational aims [15]. It refers to learning that is guided by meta-cognition (thinking about one's thinking), strategic action (planning, monitoring and evaluating personal progress against a standard), and motivation to learn [16]–[20]. A self-regulated learner "monitors, directs and regulates actions toward goals of information acquisition, expanding expertise and self-improvement" [21].

Self-regulated learners are cognizant of their academic strengths and weaknesses, and they have a repertoire of strategies they appropriately apply to tackle the day-to-day challenges of academic tasks. These learners hold incremental beliefs about intelligence (as opposed to entity, or fixed views of intelligence) and attribute their successes or failures to factors (e.g., effort expended on a task, effective use of

strategies) within their control [22]. Additionally, self-regulated learners take on challenging tasks, practice their learning, develop a deep understanding of subject matter, and exert effort towards academic success [18].

1.1.4 Problem: A-priori knowledge

Over the past years, dozens of data clustering techniques have been proposed and implemented to solve data clustering problems [23], [24]. Although these techniques have proved to be very effective and efficient, they generally depend on providing prior knowledge or information of the exact number of clusters for each dataset to be clustered and analyzed [25]. More so, when dealing with real-world datasets, it is normal not to expect or have any prior information regarding the number of naturally occurring groups in the data objects [26].

For the reasons we just mentioned, it seems obvious that in order for a practitioner or professional to succeed they must have a deeper knowledge of the different methods they might use. This includes things like: being aware of the shortcomings present in the different clustering algorithms i.e. knowing what characteristics the data must have in order for a specific algorithm to do a good clustering job, be acquainted with the available techniques to determine hyper-parameters like the number of clusters, etc.

1.2 Project proposition

This project seeks to propose a cluster analysis learning tool which allows the user to know what kinds of clustering techniques exist, how they work and which will give better results for a specific situation. To do this, the tool should rely on hands-on learning techniques, to ensure the understanding of the topics is as deep as possible; and self-regulated learning techniques, to free the user of the need of a teacher in order to understand the information presented by this tool.

Throughout the proposal, there will be mention of different clustering algorithms. A taxonomy for these clustering techniques will be presented. This taxonomy will later be used to select distinct algorithms. This selection will aim to representing as many different techniques as possible in the most meaningful way. The selected algorithms will be explained in detail to give an understanding on how they work and how they can be implemented in R. In order to reinforce the presented taxonomy, these explanations will try to expose the core principles behind each technique.

There are two main reasons behind the use of the R language to carry out this project. On the one hand, the R programming language is extensively used in data science being the second most used language, only bested by Python. On the other hand, thanks to the fact that the R programming language is dynamically typed, interpreted and has syntax similar to that of the most common languages (C, C++, Java, Python, etc.) it is really easy to learn and is often used for this very purpose. Given these reasons, the R programming language is considered a good choice to develop such learning tool.

1.3 Objectives

1.3.1 General objectives

The objective of this project is to develop a R package which implements several different clustering algorithms to give a good overview of the existing clustering techniques, how they compare to each other and what they do specifically to cluster data. Additionally, the package must work as a tool to better

explain and understand the inner workings of the algorithms it implements i.e. it must help the process of learning clustering techniques.

1.3.2 Specific objectives

To achieve the previously mentioned objective, the following requirements are proposed:

- Explore the different clustering techniques used in the state-of-the-art and propose a taxonomy.
- Make a selection from the most common techniques, making sure to pick algorithms from different categories i.e. algorithms which solve the clustering problem with completely distinct methods.
- Write a R package which implements the selected techniques.
- Whenever possible prioritize ease of understanding over computational cost. Ideally, the code should be easily modifiable by anyone using this package.
- Have the implementations explain (if asked) the step-by-step process the algorithm follows when clustering a specific dataset.
- Generate toy datasets with which to test the different implementations.
- Document the methods implemented in the package, providing an explanation on how the algorithms work.

1.4 Structure of the document

This document is divided in five chapters. These chapters can be summarized as:

- In Chapter 1 this project is introduced, the reasons behind its development are evaluated. Among these we can find the massive amount of data generated by the internet and the benefits provided by learning techniques such as hands-on and self-regulated learning. The problem to solve is presented as well as the objectives of the project, both general and specific, and the structure of the document is explained.
- In Chapter 2 context for the current state of the clustering field of study is given. The concept of clustering is explained, a taxonomy of clustering algorithms is presented, and several of the categories mentioned in the taxonomy are explained in further detail, often giving examples of algorithms following such structure.
- In Chapter 3 the development of the project is contextualized. A typical workflow for developing R packages is presented, the implemented algorithms are defined in detail, notes on how the actual implementations differ from the algorithms are given, and the way in which the package helps the user learn the inner workings of the chosen clustering techniques is explained.
- In Chapter 4 a way to test the implementations is presented. The datasets implemented in the package are introduced, the pros and cons of the algorithms are explained and showcased using the aforementioned datasets, and the explanations the implementations provide are shown.
- In Chapter 5 a summary of the development of this project is provided, drawing conclusions from this process, and some of the possible lines of work to be explored in the future are presented.

Chapter 2

State of the Art

Several algorithms have been proposed in the literature to perform clustering. In this chapter, we will present some of the most relevant ones as presented in [27] by rewriting and summarizing parts of said article. The algorithms will be divided into two main categories: partitional clustering and hierarchical clustering [28]–[39]. The former is subdivided into Hard/Crisp, Mixture resolving and Fuzzy clustering while the latter is subdivided into agglomerative and divisive clustering. Additionally the Hard/Crisp method is subdivided into Square-error, Model-based, Graph-theoretic, Density-based, Subspace-based, Search-based and Miscellaneous. A complete taxonomy for these clustering methods is shown in Figure 2.1. Note that, since they do not play a big role in the development of the package, fuzzy and miscellaneous clustering techniques will not be introduced in this chapter.

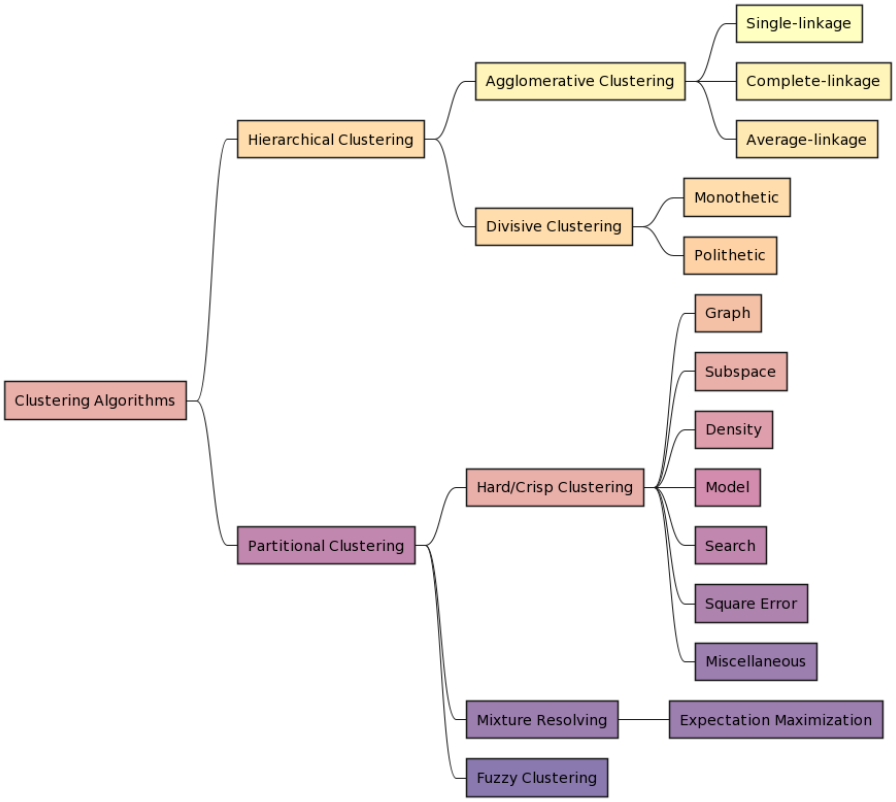


Figure 2.1: Taxonomy of clustering algorithms [27]

2.1 Partitional Clustering

In partitional clustering algorithms, data is organized into a sequence of groups with no hierarchical structure whatsoever [40], [41]. [42] says that the partitioning method is best suited for clustering problems with large datasets for which the construction of a dendrogram is too expensive in terms of computational cost. Figure 2.2 illustrates the clustering pattern representation of the partitional clustering method.

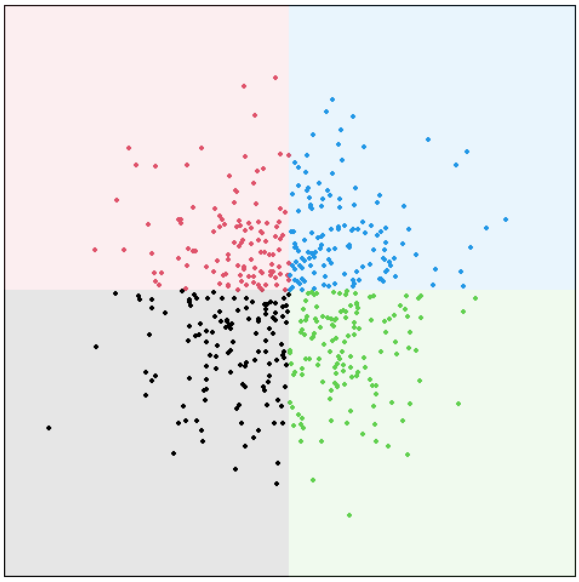


Figure 2.2: A partition with $n = 500$ and $k = 4$

When using partitional methods, the dataset of n objects is iteratively partitioned into a predetermined k number of distinct subsets through the process of optimization of a criterion function [43]. One of the most commonly used criteria is the squared error criterion. The general objective is to find the partition for which a fixed number of clusters minimizes the square error. In this case, the patterns are a collection of k spherically shaped clusters represented by the deviations of the patterns from a set of k centroids. The target cost function ζ which can be minimized is given in Equation 2.1:

$$\zeta = \sum_{i=1}^{n} \|d_i - C_j\|^q \tag{2.1}$$

where the variable n denotes the number of elements in the data set, C_j is defined as the center of the j^{th} cluster and is the center nearest to the data object d_i, while q is an integer that defines the nature of the distance function (Minkowski distance where $q = 2$ is equivalent to Euclidean distance) as discussed in [43].

In some algorithms, the square error criterion makes the k generated clusters as compact and separated as possible. This criterion function is cheaper in computational terms when compared to other criteria [40]. Since algorithms based on the square error criterion can converge to local minima, altering the initial partitions may result in different output clusters, especially if the initial points are not well separated [37]. According to [40], partitional techniques are frequently used in engineering applications where single partitions are most important and appropriate for efficient representation and compression of large databases. It has also been observed that partitional algorithms are preferred in pattern recognition due to the nature of the available data [41]. The partitional clustering method is a local search technique [44]

and local convergence. Therefore the optimal global solution cannot be guaranteed [45].

Partition-based clustering is an NP-hard optimization problem where the standard approach is to find an approximate solution [46]. According to [42] "the combinatorial search of the set of possible labelings for an optimum value of a criterion function is computationally prohibitive". For this reason, partitional clustering algorithms usually run a number of times with varying initial conditions returning the best clustering output of all the runs as the optimal solution [42]. One of the major disadvantages this algorithms can have is the need for predefined user values for parameter k, which is usually nondeterministic [42], [47]. This arbitrary choice of k leads to wrong clustering outputs [38].

Partitional clustering algorithms can be categorized based on the various techniques adopted in generating the clusters and the nature of the resultant clusters produced. These include Hard/Crisp Clustering, Fuzzy clustering, and Mixture Resolving Clustering.

2.1.1 Hard/Crisp

In a Hard/Crisp clustering algorithm each data object is assigned to a single cluster. The clustering methods under this category include Square-error, Model-based, Graph-theoretic, Density-based, Subspace-based, Search-based and Miscellaneous.

2.1.1.1 Square Error

The square error clustering method assigns data points into a fixed number of clusters based on the sum of square error criterion functions. This function computes the squared differences between each data point in a group and its estimated center value. This sum may sometimes equal to zero meaning the members in the group are identical. The formula for Sum of Square Error is:

$$\text{Sum of Square Error} = \sum_{i=1}^{n} (x_i - \bar{x})^2 \tag{2.2}$$

where n represents the number of data points and x_i represents the ith data point in the group and $\bar{x}$ is the center object relative to the group. The K-Means clustering algorithm is the best know squared error-based clustering algorithm [28].

K-Means Clustering The K-Means clustering algorithm is a partitioning technique which distributes the data points into a specified number of k clusters. In order to distribute the data points an objective function which assesses the quality of a partition is optimized. This guarantees that the similarity of data points within a cluster (intra-cluster similarity) is higher compared to that of data points in different clusters (inter-cluster similarity). K-Means is a centroid-based technique i.e. each cluster is represented by a centroid. K-Means uses the mean of the data points in a cluster as the centroids of each cluster. Initially, k data points are randomly selected from a set of the existing data points as the centroids of the k clusters. Then, the pairwise Euclidean distance between all data points and each centroid is computed. From this distances each data point is assigned to the cluster with the closest centroid. The centroids are then recomputed as the average of the data points in its cluster. This last two steps are repeated several times until stability is achieved.

As we previously mentioned, to partition the data points K-Means optimizes an objective function. This objective function is the sum of squares function. The minimization of this function for Euclidean

distances produces compact and well-separated clusters [31], [48], [49]. The sum of square function for the Euclidean distances for the K-Means algorithm is given as:

$$d_{ik} = \sum_{j=1}^{m} (x_{ij} - c_{kj})^2 \tag{2.3}$$

where d_{ik} is the Euclidean distance, x_{ij} is the jth data point for ith cluster and c_{kj} is the centroid for the jth cluster.

Among the problems with K-Means, one of the worst is that of the initial definition of the number of clusters k, since no efficient and universal method which solves this problem exists. K-Means is very sensitive to the initial centroid selection to the point where may they be wrongfully chosen, a suboptimal solution may be produced [50]. This also means that there is no guarantee that the algorithm will converge to the global optimum. Another problem with K-Means is that, given that the centroids are computed as the mean of the data points in the cluster, the K-Means algorithm's application is limited to data objects with numerical variables [28]. The K-Means algorithm is sensitive to outliers [36]. It works on the assumption that the variance of the distribution of each attribute is spherical and thus produces a roughly equal number of observations. Additionally, it is expensive in terms of memory and the number of iterations needed for the algorithm to converge is undetermined. Despite all this problems, the K-Means algorithm is still popular and widely used today [31]. This is due to the simplicity of implementation and its low computation complexity [41].

K-Means is arguably the most popular clustering method but is plagued with drawbacks such as poor scalability, sensitivity to initialization and outliers, assumed knowledge of cluster count, and local production rather than the global optimum. There are some extensions on the K-Means algorithm which seek to advance the state-of-the-art in addressing these issues. For example, the G-means [51] and the X-means algorithms [52].

2.1.1.2 Model-based Clustering

Model-based clustering makes the assumption that there is some underlying probability distribution or model the data follows [53], where clusters are represented by each of the components of the distribution. The principle of model-based clustering is to recover the underlying model and build clusters of similar data points using it to determine which data points satisfy said model. Model-Based clustering tries to optimize a predefined models fitness concerning the given data. Since clusters are generated using the given data, the number of clusters can be automatically found thus making it easy to identify outliers. In Model-based clustering, a mixture model is used in representing data, and the components of the model correspond to the different clusters.

[53] reported two ways to formulate models for cluster composition: the classification likelihood approach and the mixture likelihood approach. Model parameters can be calculated by means of the the Bayesian Information Criterion (BIC) [53]–[55] or the Maximum Likelihood Estimation (MLE) criterion [45]. The BIC can also determine between two clusters which is most likely to contain a data point [56]. There are two major approaches using this method: the statistical approach and the neural network approach [45]. Examples of the Model-Based Clustering method includes Expectation Maximization (EM) [53], [57], [58], SOM and COBWEB. A parametric mixture distribution for a random vector A can be written as:

$$f(\theta) = \sum_{b=1}^{B} \pi_b f_b \left(a|\theta_b\right) \tag{2.4}$$

where $\pi_b > 0$; such that $\sum_{b=1}^{B} \pi_b = 1$ are regarded as the mixing proportions. The $f_b(a|\theta_b)$ is the bth component density with the parameter vectors of the distribution represented as $\theta = (\pi, \theta_1, \ldots, \theta_b)$ with $\pi = (\pi_1, \ldots, \pi_b)$. The $f(\theta)$ is called the B-component finite mixture density. $f_1(\theta_1), \ldots, f_b(\theta_1)$ represent the distribution components, that is, the clusters of the parametric mixture distribution. The distribution components are the same type for all the b. A knowledge-based clustering scheme was proposed by [59] introducing the notion of conceptual cohesiveness as a precise enough way to be adopted for semantic grouping of related objects based on cohesion forests. In order to give meaning to the generated clusters, the authors presented a set of axioms that should be satisfied.

2.1.1.3 Graph-theoretic Clustering

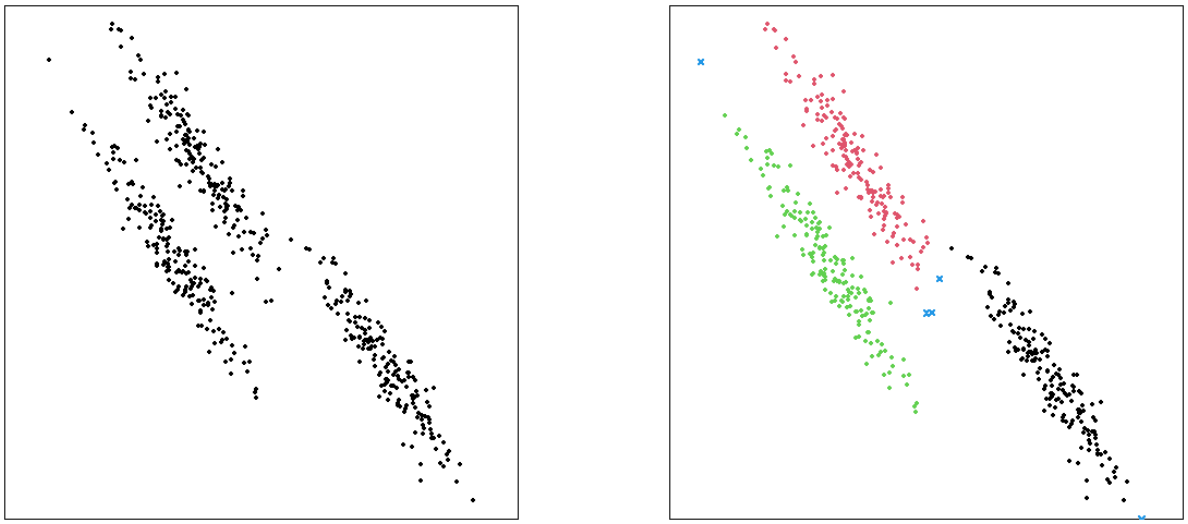
A graph is a structure amounting to a set of objects in which some pairs of the objects are in some sense related. The objects correspond to mathematical abstractions called vertices, nodes or points; and each of the related pairs of vertices are is called an edge, link or line [60]. Graphs can be used to represent feature relationships between data objects and list relevant features. In graph-theoretic clustering, clusters are represented by graphs [36] where the nodes correspond to the different data points which are connected by edges reflecting the proximity between pairs of data points [28]. The criterion function seeks to make the edge density across clusters smaller than the edge density within clusters [36]. If an edge has a weight substantially larger than that of its nearby edges, it is said to be inconsistent. Nodes are grouped into clusters based on the graph topology so that the resulting clusters have high intra-connectivity and low inter-connectivity among the generated clusters. While representing clusters in graphs is convenient, it does not help when handling outliers.

Graph theory can be used to obtain both hierarchical and non-hierarchical clusters. Graph methods which directly deal with connectivity graphs can be used in linkage metrics-based hierarchical clustering as long as the $N \times N$ connectivity matrix is sparse [40], [61]. This clustering method uses the topological properties of a network of data objects to build clusters. Finding the maximally connected sub-graphs in a graph is equivalent to the single linkage hierarchical clustering. Likewise, finding maximally complete sub-graphs in a graph is the same as the complete linkage hierarchical clustering [40]. The K Nearest Neighbor (K-NN) graph model was used to develop Chameleon (an agglomerative hierarchical clustering algorithm) [28]. Using the K-NN graph approach, Chameleon constructs a sparse graph where the weight of each edge indicates the similarity (distance) between the corresponding vertices. The K-NN graph is partitioned into several sub-clusters using a graph partitioning algorithm in order to minimize the weight of the edges to be cut. The clustering process eliminates the edges whose vertices are not within the k closest points concerning each other and uses an agglomerative hierarchical clustering algorithm to merge similar sub-clusters. Another graph representation of hierarchical clustering is the Delaunay Triangulation Graph (DTG) that uses a hyper-graph where more than two vertices are connected to an edge [62].

Zahn's clustering algorithm [63] is an example of graph-theoretic non-hierarchical clustering. Uneven edges in minimum spanning trees are detected and discarded in the bid of connecting components as clusters [40]. However, in order to select the proper heuristic to identify irregular edges, there is a need for prior knowledge of the cluster's shape. Cluster Identification via Connectivity Kernels (CLICK) is another such example. In CLICK, clusters are generated by performing the minimum weight division [32], [64]. The specification of suitable parameters and criterion properties is a problem to be addressed [40]. According to [40], "no theory exists for choosing among the various properties of graphs to select the best clustering method for a particular application".

2.1.1.4 Density-based Clustering

In density-based clustering, clusters are looked for in the pattern space where dense regions separated by low density regions are viewed as clusters. These high density regions (a.k.a. modes) are related to cluster centers, while the objects in the sparse areas separating the clusters are considered to be noise or outliers [46]. Figure 2.3 presents the clustering pattern of the density-based clustering method.



(a) Initial dataset

(b) Cluster assignation

Figure 2.3: Density-based clusterization example

In density-based algorithms, a histogram is constructed by dividing the space into non-overlapping regions which help find the modes. For each of these regions the frequency with which a data point falls into them is measured. The regions with high-frequency counts form the potential modes while the ones with low-frequency counts form the boundaries between the clusters. With this, the cluster centers are identified and the data points are added to clusters with the closest center. One of the major drawbacks of using a histogram to measure the density function is that the pattern space must be large enough to identify the sections correctly [40]. Additionally, clusters that are small in size are usually very noisy since their boundaries are defined loosely. On the other hand, gargantuan clusters have loosely defined cluster properties because of the member patterns' varied properties. It is also troublesome to locate the precise values for the histogram's peaks and valleys.

Several works proposing the general concept for mode identification has been reported in the literature [40]. This clustering method has been used extensively in engineering, mostly in remote sensing applications [65]. In some other cases, clusters are formed based on the data points density within a region. Data points are added to the cluster until the neighborhood's density is less than a given threshold. In this case, a cluster in the neighborhood of a given radius must contain a minimum number of objects concerning the specified threshold. Generation of the cluster this way enables the building of clusters with arbitrary shapes. Outliers or noisy data points are naturally eliminated. Examples include Density Based Spatial Clustering of Applications with Noise (DBSCAN), Ordering Points To Identify the Clustering Structure (OPTICS), Density-based Clustering (DENCLUE). The DBSCAN has a well-defined cluster model with fairly low complexity [46]. OPTICS solved the DBSCAN's problem of choosing an appropriate value for the range parameter producing a hierarchical output similar to linkage clustering [46].

The use of the spatial index in finding data point's neighborhood has been reported as improving the complexity of the model from $O(n^2)$ to $O(n \log n)$ compare with other methods [37]. The density-based

clustering method is reported as resistant to outliers, insensitive to data object ordering, ability to form arbitrary shape clusters, and no need for pre-stating the number of clusters [45]. However, they are not ideal for large data sets due to dimensionality. The low-density areas as noise make the algorithms based on this clustering method unable to detect intrinsic cluster structure common in real-life data. There is also the problem of cluster border detection because there is the need to have data point's density drop to show the demarcation between clusters [46].

2.1.1.5 Subspace Clustering

Subspace clustering is an extension of the traditional clustering algorithm where the clusters are searched for in the different subspaces in which a dataset exists. When it comes to large dimensional datasets, it is often better to describe them through the subspaces they exist in rather than using them as a whole [66]. In this way, the subspace clustering technique helps discover hidden knowledge in datasets with high dimensionality. Clusters existing in multiple overlapping subspaces are easily identifiable through subspace clustering. Subspace clustering removes redundant and irrelevant dimensions by means of feature selection, leaving only relevant dimensions that the clustering algorithm will use to find the clusters.

Subspace clustering algorithms are categorized into two subsections: the top-down and bottom-up approaches. The bottom-up subspace method uses an *a priori* style approach to leverage the density's downward closure property to reduce the search space. The density's downward closure property holds that dense units found in a k-dimensional subspace S, will also be found in any $(k-1)$-dimensional projection of S. Based on this, the bottom-up method creates a histogram for each dimension and selects dimensions whose density is above a given threshold. Examples of bottom-up subspace clustering include (but are not limited to) CLIQUE [67], ENCLUS [68], MAFIA [69], CBF [70], CLTree [68] and DOC [71] are examples of clustering methods that use this subspace clustering approach.

In the top-down approach, an initial approximation of the clusters in the whole feature space with equally weighted dimensions is first found. In the next step, a weight is assigned for each subspace size in each cluster using a sampling technique to improve the algorithm's performance. There is a need to specify the subspace sizes and the number of clusters beforehand, which is the first bottleneck of this approach. Dealing with outliers in the dataset is yet another bottleneck. Parameter tuning must be performed to achieve a meaningful result. Examples of top-down subspace clustering include (but are not limited to) PROCLUS [72], ORCLUS [73], FINDIT [74], COSA [75] and δ-Clusters [76].

2.1.1.6 Search-based Clustering

Traditional algorithms tend to have an additional burden placed on them as they need to be provided with *a priori* information on the number of clusters [31]. A fundamental problem of clusterization is that of determining the best estimate of cluster number. This is usually called the "automatic clustering problem" [77]. This is more pronounced when working on real-world data as these datasets are characterized by their high density and dimensionality. The lack of domain knowledge beforehand makes it difficult to figure cluster numbers, especially if the data has high dimensionality and its clusters vary in shape, size and density while sometimes overlapping. In such scenarios it is astoundingly difficult to determine the optimal number of clusters, and so is providing such information to a data clustering algorithm. Search-based clustering algorithms emerge as a solution to the need to provide the traditional clustering algorithms with this vital information [31]. They are nature-inspired metaheuristic approaches, where the structure and number of clusters in a dataset is spontaneously determined without prior information on the dataset's attributes values [78].

Automatic clustering techniques where such a requirement is out of the question are a better suited for real-world data sets such as those we previously mentioned. Automatic clustering algorithms produce the same results as their traditional counterparts without the need of any background information on the datasets [31], [40], [78], [79]. These algorithms have also been found to be able to handle automatic classification and identification of unlabeled data points in real-world datasets. Unlike traditional clustering algorithms which are mostly local search algorithms whose solutions are greatly influenced by the initial conditions, automatic clustering algorithms have a higher probability of reaching optimal global solutions. In contrast with traditional clustering algorithms, the nature-inspired clustering algorithms have demonstrated more flexibility in handling clustering problems rather than being mostly problem-specific and lacking continuity [80]. Automatic clustering algorithms treat clustering problems as optimization problems with a focus on the minimization of intra-cluster distance and maximization of inter-cluster distance [31], [81].

On one hand, finding the optimal solution of a cluster analysis problem is classified as an NP-hard problem whenever $k > 3$ (the number of clusters is greater than three) [82]. This means that, for moderately sized problems, clustering tasks could be computationally prohibitive. On the other hand, nature-inspired metaheuristic algorithms designed to be able to handle high-dimensional, complex real-world data clustering problems [83]. Moreover, their higher heuristic search capability makes them, while balancing intensification and diversification in the search, look for the most promising solution. Therefore, since most metaheuristic techniques are cheap in terms of computing power needed, they are suitable for solving clusterization problems [81], and so they became the most applied techniques for implementing automatic clustering algorithms [77]. These nature-inspired meta-heuristic algorithms have solved a wide range of continuous and discrete combinatorial optimization problems, particularly the Genetic Algorithm (GA), Differential Evolution (DE), Particle Swarm Optimization (PSO), Firefly Algorithm (FA) and Invasive Weed Optimization (IWO) [31]. Automatic clustering algorithms are far superior in performance compared with their traditional counterparts in terms of convergence speed (speed to reach a solution) and their ability to produce good quality solutions.

Some of the search-based nature-inspired techniques that have been tested as clustering algorithms include GA [40], [84], [85], DE [86], [87], Artificial Bee Colony Optimization Algorithm (ABC) [81], [88], Ant Colony Optimization (ACO), PSO [89], [90], IWO [91], Symbiotic Organisms Search (SOS), Bacterial Evolutionary Algorithm (BEA) [92], Variable Neighborhood Search (VNS), FA [93] and Tabu Search (TS).

The metaheuristics-based clustering algorithms can be classified into two: Evolutionary and Swarm Intelligence algorithms. The GA and DE algorithms come under the former group, while the rest fall under the latter. These two groups have several common design steps: they start by randomly initializing the population, then evaluating said population to identify suitable population members which represent the solution [83], from which a new population will be generated by modifying the individual-specific variation operators. The second and third steps are repeated iteratively, updating which candidate individual is best fitted in terms of the defined objective function of the problem. We will now expand on GA, ACO, ABC and PSO as they are the most commonly used algorithms under this category.

Genetic Algorithm-Based Clustering Techniques The Genetic Algorithm is a single objective evolutionary computation method developed by [94] which has been used for automatic clustering. The inspiration for this algorithm comes from Charles Darwin's principle of evolution by natural selection, constructing a robust search algorithm with minimal problem information [95]. The algorithm performs a search in large, complex multimodal landscapes and in the process obtains a near-optimal solution for the fitness function. In GA-based clustering techniques, the characteristics of GAs are applied to figure

out the proper number of clusters through evolution, providing appropriate clustering [95].

The search space parameters are represented by means of strings called chromosomes which encode a combination of cluster centroids. A collection of chromosomes forms the algorithm's population. Initially, a random population representing different search space solutions is created. Additionally, there is an objective or fitness function which measures how good the solutions represented by each chromosome are. In accordance to the principle of survival of the fittest, the chromosomes representing the best solutions are selected to 'give birth' to the next generation of chromosomes by means of two biologically inspired breeding operators: the crossover and mutation operators. The selection and breeding operations are iteratively repeated until a stopping criterion is met (e.g. until a given number of generations has been bred, a fitness score has been reached, etc.) [96]. In GA-based clustering techniques, the selection controls the search direction, while the breeding generates expands the search into new regions.

There have been several attempts at developing GA-based clustering algorithms. [97] researched the capabilities of GAs in the context of clustering problems. [98] proposed Genetic K-Means Algorithm (GKA), a GA to find a globally optimized partition of a given dataset into a specified number of clusters which uses a one-step K-Means to accelerate crossover operations. GKA searches faster than K-Means and converges to the global optimum, thus minimizing Total Within Cluster Variation (TWCV) [98]. [99] developed Fast Genetic K-Means Algorithm (FGKA), a variation of GKA featuring several improvements over it. Other GA-based clustering algorithms include Incremental Genetic K-means Algorithm (IGKA) [100], GA-clustering [101] and Genetically Guided Algorithm (GGA) [102]. [31], [77], [95] provide other references on further works on GA-based clustering. [77] discussed four categories of GA-based automatic clustering algorithms based on their chromosome encoding scheme: the centroid-based encoding of variable length, the centroid-based encoding of fixed length, the label-based encoding, and the binary-based encoding.

Ant Colony Optimization Clustering Algorithm Swarm Intelligence (SI) is a paradigm in artificial intelligence which takes inspiration from the emergent behavior in decentralized, self-organized systems. SI techniques try to imitate this emergent behavior applying it to the search of solutions in hard computational problems. They generally have a simple design, good scalability and are robust. The ACO algorithm is a stochastic metaheuristic, classified under SI, for combinatorial optimization [103]. The Ant-based clustering methods are directly modeled on ant's social behavior [103]. It is the most popular kind of SI clustering algorithm. There are two major approaches to this kind of clustering: those that are tightly bound to ants' behavioral nature and those whose loosely follow this nature. The former considers the gathering and occasional sorting of items observed in the nest and brood care of ants [104]. This behavior is directly imitated in the clustering of abstract data where the clustering objective has to be defined implicitly. The latter, which is loosely inspired by nature, handles clustering tasks as an optimization task using the ant-based optimization method to generate near-optimal clusters. This approach allows the explicit specification of the objective function, therefore offering a better understanding and prediction of the clustering performance. The ACO clustering algorithm falls under this second group. It is inspired by the foraging behavior of mass recruiting ants. The ants use pheromones to mark areas of promising forage and potential food sources [103], [105]. [106], [107] carried out some research on the ACO-based clustering algorithms. An ant-based clustering algorithm was also presented by [108], which finds the adequate number of clusters and initializes the Fuzzy C-Means algorithm. [109] worked on adaptive time-dependent transporter ant for clustering.

Particle Swarm Optimization The PSO is another general-purpose optimization metaheuristic. PSO is inspired by unsophisticated agents that interact locally among the neighboring individuals and their

environment. The collective behaviour of these is complex enough that it is actually valuable for solving optimization problems [77], [110]. PSO was introduced as a population-based search algorithm where there is a swarm which is a population of particles each of which represents a complete solution. During the optimization process, this swarm of particles moves cooperatively in the region defined as the objective function. The particles move according to a set of forces directed towards good positions in the search space which has already been explored by the swarm. According to [77], "the particles explore the search space by adjusting their trajectories iteratively according to self-experience and neighboring particles". Initially the swarm has several particles placed at random positions in the search space with randomly assigned velocities. At every iteration, each particle will evaluate the objective function at its position, updating its position, velocity, and memory for its individual best position [111].

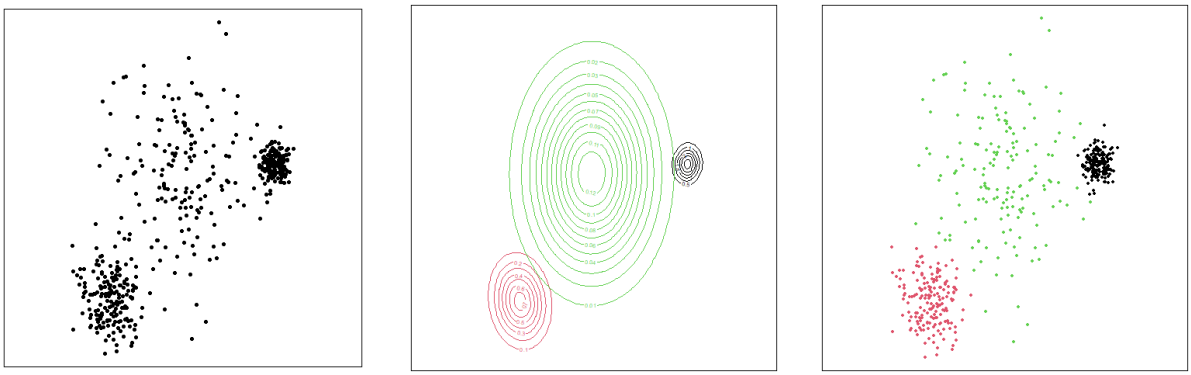
The first applications of PSO for solving clustering problems used a fixed number of clusters. PSO was given a new use in searching for the clusters' optimal centroids assigning the data points to whichever was the closest centroid. This applications were further extended in the presentation of the Dynamic Clustering Particle Swarm Optimization (DCPSO) [77], [112]. Another PSO-based segmentation algorithm for automatically grouping image pixels into different regions was proposed by [113]. Other researches on PSO-based clustering algorithms can be found in [81], [114]–[121]. It has been observed that PSO-based clustering algorithms give excellent results in quality clustering to find the correct cluster number. The PSO basic algorithmic form is characterized by extreme simplicity. It is mostly used to optimize the functions of continuous variables.

2.1.2 Mixture Resolving Algorithms

The Mixture Resolving Algorithm or mixture-based algorithm makes the assumption that a set of data points emanates from a mixture of instances of multiple probabilistic clusters. This means that, to generate the data points, a probabilistic cluster was chosen according to the cluster's probabilities and with it a sample was generated (following its probability density function) and added to the set. Therefore, when clustering, the data set is assumed to be a mixture of different cluster groups with varying proportions. The mixture likelihood-based approach to clustering is model-based since there is a beforehand requirement for the specification of each component density of observation. [122] stated that a statistical model to be used must be stated or known ahead in the clustering of samples from a population. Given the relationship between these two algorithms, it is possible to conduct estimation analysis and hypothesis testing of clustering methods based on mixture models using standard statistical theory. [123], in support of this, had stated that the mixture likelihood-based approach "is about the only clustering technique that is entirely satisfactory from the mathematical point of view". It makes use of well-defined mathematical models, investigating them with well-defined statistical techniques and gives a measure of significance for the results. Furthermore, determining what the best number of clusters is can be easily done in a Mixture-based algorithm as it has a clear probabilistic foundation. [124] stated that providing an effective clustering of various data sets under various experimental designs is one of the mixture model's usefulness. However, very strong assumptions are made on the data distribution; clusters are represented by a single simple distribution, limiting the cluster's shape [125]; and the algorithm is computationally expensive. Figure 2.4 shows an clusterization example with this algorithm.

2.1.2.1 Expectation Maximization

The Expectation Maximization (EM) algorithm is a framework that employs two major steps in approaching the maximum likelihood of estimates of parameters in a statistical model: the Expectation



(a) Initial dataset (b) Mixture model (c) Cluster assignment

Figure 2.4: Mixture resolving clusterization example

Step (or E-step) and the Maximization Step (or M-step). Initially, the probabilistic distribution parameters (such as the mean and standard deviation) are selected e.g. the values are randomly assigned. In the E-step, the data points are assigned to clusters based on the probabilistic clusters' parameters i.e. each object's likelihood to belong to each distribution is computed. In the M-step, a new clustering that maximizes the expected likelihood is found i.e. the probabilistic distribution parameters are adjusted to maximize each cluster object's expected likelihood. The E-step and the M-step are iteratively executed until the probabilistic distribution parameters converge i.e. the change is null or almost null. Thus, each iteration of the EM algorithm requires several computations. The product of the number of data points and the number of mixture components scales linearly with this iterative computation, limiting the applicability of this algorithm in large-scale applications [126]. Furthermore, the algorithm's output is highly dependant on the initial parameters. Nevertheless, the EM algorithm is easy to implement, and there is no need to set any parameters that will influence the optimization algorithm [126].

2.2 Hierarchical Clustering

The hierarchical clustering method partitions data into levels, building a hierarchy. Clusters are iteratively merged or split to generate a dendrogram depicting the formulated clusters' hierarchical structure [36]. This clustering method allows exploring data on different levels of granularity. A dendrogram representation for the hierarchical clustering method is displayed in Figure 2.5.

The hierarchical clustering algorithms can easily handle any similarity measure and flexible level of granularity [50]. As a result, they apply to any attribute type. However, these algorithms can be afflicted with irreversible splits or merges such that wrongly constructed clusters cannot be revisited. Additionally, their application in large-scale datasets is limited because it has high computational complexity. On that note, most of the hierarchical clustering has at least a computational complexity of $O\left(N^2\right)$ [36]. Apart from these, there is also the problem of the vagueness of termination criteria and lack of robustness due to its sensitivity to noise and outliers. It has also been reported that algorithms which rely on the use of euclidean distances, tend to form spherical shapes. One last drawback, which affects agglomerative methods, is that those that are practical in terms of time efficiency require memory usage proportional to the square of the number of groups in the initial partition.

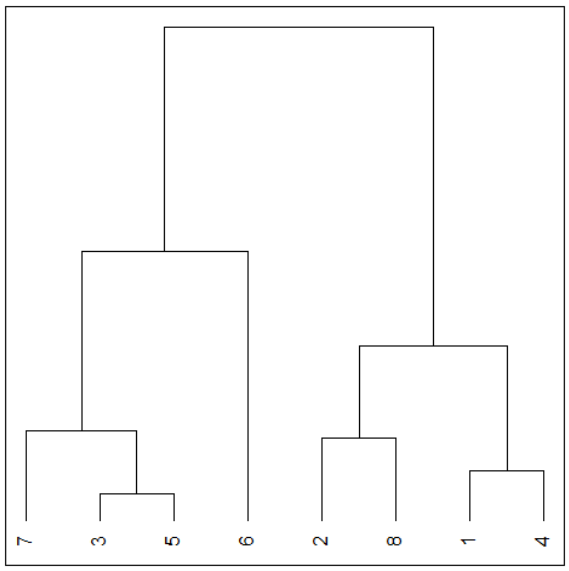


Figure 2.5: Dendrogram representation for the hierarchical clustering of eight objects

2.2.1 Agglomerative Clustering

In the agglomerative clustering method, clusters are constructed from N clusters containing a single data point each, which are iteratively merged into larger clusters that form the hierarchy's various levels until the entire object includes a single cluster or until the stopping criterion is met. The final cluster, containing all data points is the root of the hierarchy. When merging, the two closest objects, according to the similarity measure, are the ones to be combined. It requires at most, $N-1$ iterations to complete the clustering operation since every iteration merges two clusters into a single one, reducing the total amount of clusters by one.

The merging of clusters is based on a proximity measure or linkage metric that generalizes the distance between individual points to the distance between subsets of points. This method utilizes a $N \times N$ similarity matrix with which the linkage metrics used for the clustering are constructed. The construction of this matrix is achieved by finding the similarity between each pair of data points. From this matrix, the linkage criterion can be calculated by finding the pairwise distance between clusters. There are three basic linkage metrics: single-linkage, complete-linkage and average-linkage [36], [42], [127], [128].

The single-linkage metric, a.k.a. nearest neighbor, minimum or connectedness metric, measures the nearest distance from any member of one cluster to any other cluster member. It has a chaining effect with the tendency of producing elongated clusters [50].

The complete-linkage metric, a.k.a. the maximum, diameter or farthest neighbor metric, determines the distance between two clusters by measuring the longest distance from any member of one cluster to any member of the other cluster. Its clusters are more compact and tightly bound than single-linkage clustering [40].

The average-linkage metric is also regarded as the minimum-variance linkage [36], [40]. It finds the mean or median of the distances among all the data points between clusters [50].

These three linkage metrics consider all the points of a pair of clusters. [129]–[131] implemented SLINK, CLINK and the Vorhees' method which are the implementations of the single-linkage, complete-linkage and average-linkage hierarchical clustering algorithms, respectively.

Ward's clustering method [132] implements an agglomerative clustering algorithm based on K-Means' objective function where the criterion for choosing the pair of clusters to merge at each step is based on the optimal value of an objective function. Furthermore, it is considered a general agglomerative hierarchical clustering procedure. This clustering method is most appropriate for quantitative variables and not binary ones. [133] developed a non-parametric hierarchical, agglomerative clustering algorithm based on the use of a conventional nearest neighbor to determine the Mutual Neighborhood Value (MNV) and Mutual Nearest Neighbors (MNN) of a sample point. Their simple, non-deterministic and non-iterative algorithm requires low storage and can discern non-spherical and spherical clusters. More so, this method was reported to have the ability to discern mutually homogeneous clusters and applications to a wide class of data of arbitrary shape, large size, and high dimensionality.

Clustering Using Representative (CURE) is an implementation of the agglomerative hierarchical clustering algorithm intended for large databases. Outliers do not have that much effect on it and it can identify clusters of different shapes and sizes although with less cluster quality than those of Balanced Iterative Reducing and Clustering Using Hierarchies (BIRCH). It has a time complexity of $O\left(N^2 \log N\right)$ and its performance is particularly good on 2-dimensional datasets. CURE solves the scalability problem with the use of data sampling and partitioning; clusters with fine granularity are first constructed in partitions. Clusters are represented by a fixed number of points scattered around it. The distance between two clusters is generated by finding the minimum distance between the two clusters' representative points. The use of scattered representative points enables CURE to identify clusters of diverse sizes and shapes. The scattered representative points are shrunk to the cluster's geometric centroid as the clustering progresses based on the user-specified factor. The choice of the input parameters for CURE: the shrink factor, representative point number, sample size, and the number of partitions affect the clustering output. CURE was developed to work on datasets with numerical attributes [61].

2.2.2 Divisive Clustering

The divisive hierarchical clustering algorithm follows a process opposite of that of the agglomerative clustering, beginning with every object in a single cluster, dividing every cluster into smaller chunks, until the required number of clusters is attained [134]–[136]. The standard method of cluster splitting is to consider every possible bipartition before splitting them into two clusters containing one or more elements. It should be obvious that the full enumeration process offers a universal optimum at the cost of being very expensive in terms of computation. For this reason, there have been investigations on various divisive clustering approaches which do not consider all bipartitions. For example, a method that uses K-means to split the clusters in order to obtain better results than the traditional K-Means or agglomerative method was proposed by [137]. [138] proposed an Improved particle optimizer (IDPSO) to determine the closest optimal partition hyperplane for splitting designated clusters into two smaller ones, which is both practical and efficient. In another study, [139] investigated a method called Reference-point-based dissimilarity measure (DIVFRP) combining it with the purpose of divisive clustering: dataset partition. [140] and [141] used an average dissimilarity between an object and a set of objects to investigate the iterative divisive procedure.

Divisive clustering can be divided into monothetic and polythetic methods. Monothetic divisive clustering employs a single variable in each splitting, by separating objects with specific values from those without value [142]. Monothetic is usually a variant of the association analysis method [143] and is proposed for binary data. Various studies have applied monothetic clusters for problem-solving. For instance, [144], [145] employed a monothetic clustering approach on interval and histogram data. Similarly, [146] utilized the monothetic clustering method on multi-modal data. However, those monothetic approaches

decrease the number of computations required to identify an optimum bipartition, such that only $p\,(n-1)$ bipartitions are needed for testing to determine the optimum bipartition instead of all $\left(2^{n-1}-1\right)$ likely bipartitions. The larger the number of objects, the possible bipartition number is further decreased by the monothetic method. Besides, they offer binary questions that facilitate the interpretation of clustering structures.

The polythetic divisive clustering approach utilizes all variables concurrently via dissimilarity or distance values. It does not rely on single variable order but depends entirely on distance values, and the distance values reflect on all variable dissimilarity concurrently [147]. However, large variables (variables with many dimensions) may lead to scalability issues. [148] proposed a modern divisive clustering algorithm termed Hierarchical grid clustering using data field (HGCUDF). In this approach, hierarchical grids divide large datasets and their subset's are clusterized. However, the clustering regions limit the search scope, minimizing the data space for producing data fields. HGCUDF exhibits rapid execution and stability, which improves the clustering results on a large automated dataset. In another study, [149] investigated a model-based clustering technique for high-dimension datasets. This technique is divided into three steps: multi-modal splitting, iterative weighted sampling, and uni-modality preserving merging to measure the model-based clustering approach of large high-dimensional datasets. This method of clustering algorithm solves the problem of small datasets and the effective scaling of the large datasets when evaluated with synthetic datasets compared with the conventional methods. It is helpful in immune response jobs and can tremendously regulate rare populations.

For the most part, the clustering algorithms in the literature are focused on binary data. However, the clustering of categorical data has attracted more researchers in recent years. Many researchers have proposed different divisive hierarchical clustering algorithms to combat this problem. For instance, [150] suggested Maximum Dependency of Attributes (MDA) for divisive hierarchical clustering attributes selection. The maximum dependency of attributes is created by relying on attributes dependency in rough set theory, which measures the dataset's attributes dependency. In another study, [151] investigated a bi-clustering method for choosing two-valued attributes by considering multi-valued attributes and a Total Roughness (TR) approach. They maintained that high TR attributes attain optimum performance and are suitable for cluster splitting. [152] developed a Min-Min-Roughness (MMR) metric to resolve the uncertainty in the categorical data clustering process. However, MMR signifies TR's reverse and does not yield clustering algorithms with comparative improvement in complexity or accuracy [150], [153]. [154] investigated a divisive method for categorical data based on Multiple Correspondence Analysis (MCA). Similarly, [155] implemented information theory-based divisive clustering for categorical data by employing Mean Gaian Ratio (MGR) to choose clustering attributes and select class equivalents on the cluster attribute using cluster entropy. Although divisive clustering is appealing based on computational time, partitioned clusters' quality is better than a divisive one.

[149] proposed a model-based Scalable Weighted Iterative Flow-clustering Technique (SWIFT) for high-dimensional large datasets. The model consists of three stages: multimodality splitting, weighted iterative sampling, and unimodality; preserving. Merging for model-based clustering scaling on high-dimensional datasets are constructed to be effectively scalable to large datasets, offering a significant enhancement when compared with the current soft clustering approaches [156], [157]. These three major SWIFT stages are motivated by two main requirements: scalability to large datasets and rare population identification. In SWIFT, multimodality splitting and weighted iterative sampling identify rare populations. This algorithm is usually met for Flow Cytometry (FC) and finding rare populations. The multimodality stage plays a vital role in identifying rare subpopulations. When evaluated with synthetic datasets, the algorithm solves small datasets and can effectively scale large datasets compared to conventional methods. SWIFT may also be employed to represent skewed clusters by LDA-based agglomerative

merging, which decreases clusters numbers as it preserves the separate unimodal populations.

The interaction between the merging and multimodality splitting in many clusters uses a reasonable heuristic (cluster modality). It is more reasonable when compared with knee point in entropy plots formally employed [156], [158]. The algorithm advantageous immune response tasks and efficient scaling on large FC datasets. Moreover, the soft clustering method utilized in SWIFT is essential for understanding the overlapping clusters compared to the complex clusters approach like K-Means [159] or spectral clusters [160]. SWIFT has the power to control the tremendously rare populations. SWIFT is partially synonymous with flowPeaks [157] since both depend on unimodality criterion. Thus, flowPeaks focuses on the significant peaks without modality splitting and leans in missing tiny overlapping clusters. Consequently, one of the limitations of SWIFT is that it is restricted to a specific clustering task [149].

2.2.3 Implementations that Improve on Hierarchical Clustering

Over time, several improvements have been made on the traditional algorithm in order to overcome the deficiencies of hierarchical methods. One enhancement that takes hierarchical clustering' scalability limitation (performance on large datasets) into consideration is the Balanced Iterative Reducing and Clustering Using Hierarchies (BIRCH) clustering algorithm [161]. BIRCH is based on the use of Cluster Features (CF). CF is a triple that summarizes the information maintained about a cluster. This triple contains the cluster objects total number n, the linear sum of attribute values of the cluster objects LS, and the sum of squares of the attribute's values of the cluster object SS, $CF(n, LS, SS)$ [38]. The CF triples are kept in a tree form, and only the tuples are kept in the main memory.

The BIRCH algorithm consists of 4 phases, the 2nd and 4th of which are not mandatory [38]. Phase 1 handles the scanning of the entire dataset and the construction of the CF tree. The information stored in this tree is prepared carefully in order to properly reflect as much information as possible in the dataset while adjusting to the limitation imposed by the limited amount of memory with crowded data points grouped as fine sub-clusters. During this phase, outliers are treated as sparse data points and are removed from the dataset. This process ensures no other input–output operation will be required in the following phases, thus reducing the computation time for the remaining steps. Additionally, the clustering is performed on smaller sub-datasets of each sub-cluster in the entries of leaves of the CF tree since these are built by incrementally updating the CF [38]. The leaf ordering in the initial tree construction produces a better data locality, enhancing the clustering output. The BIRCH algorithm has proved to be able to handle outliers, large datasets, and produce a good clustering output that is not affected by order of input data. It is reported to be computationally efficient, achieving $O(N)$ computational complexity [37]. The efficiency of is, however, dependent on proper parameter setting. It is also biased towards non-spherical clusters due to the use of diameter/radius to control the cluster boundary. Evaluation of BIRCH using both synthetic and real datasets showed that it returns a better result in computational time complexity, the robustness of the approach, and cluster quality.

Chapter 3

Implementation

In this chapter we will take a look at both how to develop R packages, how where the algorithms in the package implemented, what problems did those implementations come across, how where those problems dealt with, and how do the implementations manage to explain the step-by-step process of each algorithm. The package implements five algorithms: K-Means, DBSCAN, GMM with EM, AHC, DHC.

The aim of each of the implementations is to provide a Minimum Viable Product (MVP) i.e. functions that works fairly fast for the typical use cases of a learning environment, which is able to generate a step-by-step explanation of what is happening at any given time during the execution. The former grants that a student can experiment with any method to get an intuition for its pros and cons, while the latter allows the student to develop a deeper understanding of the methods by allowing them to peek at its working mechanism.

3.1 Developing R packages

A package is a convention for organizing files into directories. R packages commonly have five parts: `DESCRIPTION` and `NAMESPACE`, two files containing matadata i.e. the name of the package, a description, the funcions to export, etc.; `R/`, a directory containing all the source code files of the package; `tests/`, which holds tests files to verify the correction of the code; `man/` and `vignettes/`, where the documentation, tutorials and how-tos are stored; and `data/`, to include datasets within the package. While there are multiple packages useful to package development the most common one is the devtools package. This package's aim is to make package development easier by providing R functions that simplify and expedite common tasks.

All the functions provided by this package accept a path as an argument. If no path is provided, the current working directory is used instead. Therefore, whenever we are to develop a new package we can create a RStudio project with all the necessary files with `create_package()`. This project can be handled with the git version manager with `use_git()` and this repository can be uploaded to github with `use_github()`.

Once the package project is in place, the devtools functions we will be using the most are: `load_all()`, which loads the package code; `document()`, which rebuilds the documentation and the `NAMESPACE` file; `test()`, which runs the unitary tests; and `check()`, which does a full check of the package looking for any possible errors in it. These functions define a workflow where the code can be edited and used thanks to `load_all()`, if there are any tests they can check the validity of any new code with `test()`, the documentation files can be regenerated when the roxygen is modified, and the

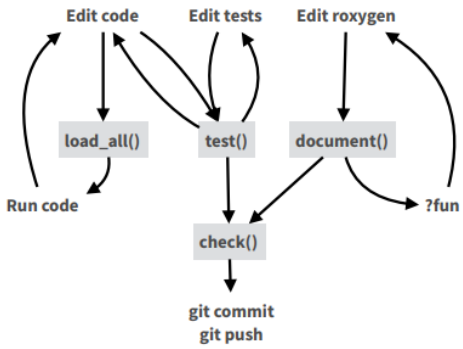


Figure 3.1: Package development workflow with devtools

package can be checked for errors with `check()` to ensure the new version can be commited to the git repository. An diagram showing this workflow can be seen in Figure 3.1.

Packages can also be tested against different systems with `check_win()` and `check_rhub()`. These are the same systems CRAN uses before submission. If all the checks have succeeded the `release()` function makes sure everything is OK with the package, then builds and uploads the package to the CRAN repositories so it is widely available.

In addition to the devtools package, there is yet another very commonly used tool when working with the R programming language: RStudio. RStudio is an Integrated Development Environment (IDE) for R. It is available in two formats: RStudio Desktop, which is a regular desktop application; and RStudio Server, a remote server with allows accessing RStudio using a web browser. The developers of RStudio also develop many packages (like the devtools package) and integrate them in their IDE, making the development faster and more comfortable to the user.

3.2 Algorithms

3.2.1 K-Means

K-Means (Algorithm 3.1) is a partitional, centroid-based clustering method which aims to partition the data points into k groups such that the sum of squares from points to the assigned cluster centers is minimized. At the minimum, all cluster centroids are at the mean of their Voronoi sets (the set of data points which are nearest to the centroid). The algorithm follows a 2 to n step process:

1. The first step can be subdivided into 3 steps:
   (a) Selection of the number of clusters k, into which the data is going to be grouped and of which the centroids will be representatives. This includes the selection of k centroids.
   (b) Computation of the pairwise distances between the data objects and the centroids.
   (c) Assignment of each observation to a cluster. The observations are assigned to the cluster represented by the centroid to which they are the closest.
2. The next steps are just like the first but the first sub-step is replaced with the computation of the new centroids. The centroid of each cluster is computed as the mean of the observations assigned to said cluster.

The algorithm stops once the centroids in step $n+1$ are identical to those of step n. However, this convergence does not always take place. For this reason, the algorithm also stops once a maximum number of iterations is reached.

Algorithm 3.1: K-Means

Input:
`data`. A matrix with n rows, one per observation.
`centers`. The number of clusters.
`max_iterations`. The maximum number of iterations.
`initialization`. The initialization method.
Output: An object containing:
`cluster`. A label for each observation indicating its cluster.
`centers`. The centroids the algorithm converged to.
`metrics`. Some metrics evaluating how good the clustering is.

```
1  begin
       // Variable initialization ...
2      centroids ← initialization(data, centers)
3      old_centroids ← null
4      iterations ← 0
5      while old_centroids ≠ centroids ∧ iterations < max_iterations do
           // Keep track of the old centers and the current iteration
6          old_centroids ← centroids
7          iterations ← iterations + 1
           // Label observations with the id of the closest centroid
8          distances ← dist(centroids, data)
9          foreach obs ∈ data do
10             cluster[obs] ← argmin(distances[, obs])
           // Compute the new centroids as the mean of the observations with
              the id of the centroid as its label
11         foreach c ∈ centroids do
12             observations ← filter(data, obs → cluster[obs] = c)
13             c ← mean(observations)
       // Label observations with the cluster they belong to
14     distances ← dist(centroids, data)
15     foreach obs ∈ data do
16         cluster[obs] ← argmin(distances[, obs])
       // Compute the metrics
17     metrics ← evaluate(data, cluster, centroids) return cluster,
        centroids, metrics
```

K-Means is very sensitive to the initial centroid selection to the point where may they be wrongfully chosen, a sub-optimal solution may be produced. To solve this problem, several initialization algorithms have been proposed in the literature. In this implementation we focus on two of them:

- **random**. A set of k observations is chosen at random from the data as the initial centers.
- **K-Means++** [162]. This algorithm guarantees to find a solution that is $O(\log k)$ competitive to the optimal K-Means solution. The algorithm can be described as a k step process:
  1. The first centroid is chosen at random from the data.
  2. The next centroid is chosen from the remaining observations with probability proportional to the square distance to the closest centroid. This step repeats until k centroids have been chosen.

Algorithm 3.2: K-Means - Random initialization

Input:
`data`. A matrix with n rows, one per observation.
`k`. The number of clusters.
Output:
`centers`. A matrix with `k` rows, each of which is an observation from `data`

```
1  begin
       // Choose k random observations from data
2      n ← rows(data)
3      centers ← data[random(n, k), ]
4      return centers
```

Algorithm 3.3: K-Means - K-Means++ initialization

Input:
`data`. A matrix with n rows, one per observation.
`centers`. The number of clusters.
Output:
`centers`. A matrix with `k` rows, each of which is an observation from `data`

```
1  begin
       // Assign an equal probability to every observation
2      foreach obs ∈ data do
3          prob[obs] ← 1 / rows(data)
4      for c = 1 to centers do
           // Choose a random observation from data according to their
              probabilities
5          centers[c, ] ← sample(data, prob)
           // Recompute the probabilities
6          distances ← dist(centers, data)
7          foreach obs ∈ data do
8              prob[obs] ← min(distances[, obs])
9          prob ← prob / sum(prob)
10     return centers
```

This algorithm was implemented in the `kmeans` function. This function is written after Algorithm 3.1. The complete source code can be found in Appendix A.1. The `kmeans` function accepts four or more arguments. The first argument, `data`, is a set of observations, presented as a matrix-like object where every row is a new observation e.g. a `matrix` or `data.frame`. The second argument, `centers`, is either the number of clusters, in which case the centroids are chosen according to the `initialization` parameter; or, unlike Algorithm 3.1, a set of initial cluster centers. The third argument, `max_iterations`, is the maximum number of iterations the algorithm is allowed to perform. The fourth argument, `initialization`, specifies the initialization method to be used. `initialization` can be one of `"random"` or `"kmeans++"`, where the latter is the default. Any additional arguments other than those four are used in any call this function does to `proxy::dist`.

Initially, the function takes these arguments and makes sure they make sense e.g. `max_iterations` is a positive integer. Then, it checks to see if the centroids have to be initialized. If they have to, there will be another round of checks to make sure the amount of centroids makes sense i.e. it is a positive integer smaller than the amount of observations. If it does, contrary to what we see in Line 2 where `initialization` is used as a function, the `kmeans` function expects `initialization` to be a string. This string is converted to a number indicating the initialization method to use. This is done with the `grep` function to allow sub-strings of the method names as valid initialization values e.g. `"km"` is equivalent to `"kmeans++"` as it is a sub-string of `"kmeans++"` not present in `"random"`. Once `initialization` is an integer one of two functions is selected and with it the centroids are finally initialized.

The `random_init` function implements the `"random"` initialization method. This method samples k random observations from the data. It is written after Algorithm 3.2. The `kmeanspp_init` function implements the `"kmeans++"` initialization method. This method makes use of a vector with the probabilities to sample any specific point as a centroid. Initially all the elements in the vector have the same probability. Then k iterations are performed in which a centroid is sampled and stored and the probabilities recomputed for every observation. This includes those which have already been chosen as centroids. However, no observation will be picked twice since the centroids are closest to themselves with a squared distance of zero, meaning their probability to be chosen will be null. It is written after Algorithm 3.3.

With the centroids initialized, the function heads into the algorithms main loop. At every iteration, the algorithm updates the number of iterations it has gone through and the centroids obtained in the previous iteration. Then, the pairwise distances between the centroids and the observations are computed. All distance computations are performed with the `proxy::dist` function. This is a fast substitute to R's default `dist` which allows computing distances between a dataset and itself as well as two different datasets. With these distances, every observation is assigned the cluster corresponding to the centroid it is closest to.

From the cluster assignments, the new centroids are be computed as the mean of the observations assigned to it. However, this operation may leave some centroids as undetermined. This is due to the fact that centroids can be assigned no observations. In this case, the centroid is simply not updated. When the new centroids have been computed, they are compared with the centroids from the previous iteration to see if they are identical. If they are, the algorithm has converged and the function exits the main loop.

Finally, the observations are assigned a cluster one last time using the distances to the latest centroids. This is the cluster assignment the funtion returns. Additionally, some metrics on the performance of the clusterization are computed. The cluster assignments, centroids and metrics are put together inside a `kmeans` structure, which is native to R, and the structure is returned.

3.2.2 DBSCAN

Density Based Spatial Clustering of Applications with Noise (DBSCAN) is a density-based clustering technique (Algorithms 3.4 - 3.5). This algorithm aims to partition a dataset into clusters such that the points in a cluster are close to each other and the points in different clusters are far away from each other. The clusters are defined as dense regions of points separated by regions of low density. This algorithm follows a two step process:

1. For each observation/point, the neighborhood of radius *eps* is computed. If the neighborhood contains at least min_{pts} points, then the point is considered a **core point**. Otherwise, the point is considered an **outlier**.
2. The second step is itself a two step process:
   (a) For each core point, if the core point is not already assigned to a cluster, a new cluster is created and the core point is assigned to it.
   (b) The neighborhood of the core point is explored. If a point in the neighbourhood is a core point then the neighborhood of that point is also explored. This process is repeated until all points in the neighborhood have been explored. If a point in the neighborhood is not already assigned to a cluster, then it is assigned to the cluster of the core point.

   Whatever points are unassigned after this step, are considered outliers.

Algorithm 3.4: Density-Based Spatial Clustering of Applications with Noise

Input:
`data`. A matrix with n rows, one per observation.
`eps`. The distance within which an observation's neighbors are found.
`min_pts`. The amount of neighbors an observation needs to have to be dense.
Output: An object containing:
`cluster`. A label for each observation indicating its cluster (or 0 if it is an outlier).

```
1 begin
    // Variable initialization ...
2   cluster[data] ← UNCLASSIFIED
3   cluster_id ← next_id(NOISE)
    // Traverse the observations in a BFS fashion ...
4   foreach obs ∈ data do
5     if cluster[obs] = UNCLASSIFIED then
        // When expanding a node, if a cluster is created update the
           cluster_id ...
6       if expand(data, obs, cluster, cluster_id, eps, min_pts) then
7         cluster_id ← next_id(NOISE)
8   return cluster
```

This algorithm was implemented in the `dbscan` function. This function is written after Algorithms 3.4 - 3.5. The complete source code can be found in Appendix A.2. The `dbscan` function accepts three or more arguments. The first argument, `data`, is a set of observations, presented as a matrix-like object e.g. a `matrix` or `data.frame`, where every row is a new observation. The second argument, `eps`, is a number determining how close two observations have to be in order to be considered neighbors. The third argument, `min_pts`, is an integer determining the minimum amount of neighbors an observation needs to have within `eps` to be considered core i.e. a dense region. Any additional arguments are passed to the `proxy::dist` function any time it is called.

Algorithm 3.5: DBSCAN - `expand` procedure

Input:
`data`. A matrix with n rows, one per observation.
`obs`. A row from `data`.
`cluster`. A label for each observation indicating its cluster.
`cluster_id`. The id of the cluster to be created.
`eps`. The distance within which an observation's neighbors are found.
`min_pts`. The amount of neighbors an observation needs to have to be dense.

Output: An object containing:
`new_cluster`. A boolean indicating whether a new cluster was created or not.

```
1  begin
       // Neighbor search ...
2      neighbors ← region_query(data, obs, eps)
3      if size(neighbors) < min_pts then
4          cluster[obs] ← NOISE
5          return False
6      cluster[neighbors] ← cluster_id
7      neighbors ← remove(neighbors, obs)
8      while size(neighbors) ≠ 0 do
9          frontier ← pop(neighbors)
10         new_neighbors ← region_query(data, frontier, eps)
11         if size(new_neighbors) < min_pts then
12             foreach neig ∈ new_neighbors do
13                 if cluster[neig] ∈ [UNCLASSIFIED, NOISE] then
14                     if cluster[neig] = UNCLASSIFIED then
15                         push(neighbors, neig)
16                     cluster[neig] ← cluster_id
17     return True
```

Initially, the algorithm labels every observation as unexplored/unclassified. Additionally, the implementation computes the pairwise distances between the observations in data. This distances are later passed to the `expand` and `query_region` sub-functions. The reason behind this is that, while the original algorithm (and the pseudo-code for that matter) assumes `data` is a structure like a *kd-tree* or a *R*-Tree* with which the neighbors within `eps` can be computed in $O(\log n)$ time, making the total execution time $O(n \log n)$ with only $O(n)$ memory. However, the implementation doesn't count with such a structure and uses the matrix of pairwise distances instead. This makes looking for the neighbors within `eps` a $O(n)$ time operation and the overall algorithm $O(n^2)$ time using $O(n^2)$ memory. While this means the implementation is really bad in comparison (in terms of computational cost), it also means it is easier to understand as it removes the need to know about these tree data structures and their inner workings.

Once the necessary variables are initialised the algorithm starts to traverse the observations. The algorithm loops over every observation and, if it has not yet been expanded, it is expanded by calling `expand` over it. The `expand` sub-function expands an observations neighborhood by traversing a graph where the nodes are the observations in `data` and there are edges directed from every dense observation to its neighbors within `eps`. This traversal is done in a Breadth First Search (BFS) fashion, which results in every node of the graph being expanded exactly once.

Since non-dense observations cannot have outward pointing edges, any call of `expand` over a non-dense observation returns `False` immediately as it cannot create new clusters. However, calls over dense observations explore the neighborhood of the observation assigning every visited observation to the same cluster and returning `True`. Since `expands` only returns `True` when it assigns observations to a cluster, the algorithm uses this as a signal to change `cluster_id` and thus assign points to a new cluster the next time an observation is expanded.

Once the loop has gone through every observation, `cluster` has a label other than `UNASSIGNED` for every observation. This labels determine to which cluster every observation belongs. Therefore, this structure is what the algorithm has to return. The `dbscan` function i.e. the R implementation of the algorithm, complements this structure with the `eps` and `min_pts` used to cluster `data` as well as the size of each cluster.

3.2.3 Gaussian Mixture

Gaussian Mixture Model with Expectation Maximization (GMM with EM) (Algorithm 3.6) is a model-based and mixture-resolving hybrid technique which aims to partition a dataset into clusters such that the dataset is represented by a probabilistic model where each component of the model represents a cluster. In the case of this specific algorithm, each cluster is represented by a Gaussian distribution. The algorithm follows a two to n step process:

1. Initially, a model is randomly generated. Since the model is a mixture of Gaussian distributions and these can be represented by their mean, covariance matrix and weight; as many of these three parameters are generated as clusters are needed.

2. The current model is optimized with the EM algorithm. This algorithm consists of two steps:

   E. During the E-Step, the probabilities of each observation belonging to each cluster are computed i.e. for every observation and cluster we compute how strongly should we Expect the observation to belong to the cluster.

M. During the M-Step, each of the models components' parameters are recalculated for them to maximize the probabilities obtained in the E-Step e.g. the mean of a component is computed as the weighted average of the observations where the weight of any observation is its probability to belong to said component (which we computed in the E-Step).

These two steps are performed iteratively until a certain amount of repetitions is hit or until the changes are sufficiently small.

Algorithm 3.6: Gaussian Mixture Expectation Maximization

Input:
`data`. A matrix with n rows, one per observation.
`k`. The number of clusters the mixture model considers.
`max_iter`. The maximum number of iterations of the EM algorithm.
Output: An object containing:
`cluster`. A label for each observation indicating its cluster (or 0 if it is an outlier).
`model`. The mixture model found by the algorithm.

```
1  begin
       // Initialize the mixture model with random parameters
2      model ← initialize(data, k)
       // The fitness of the model will be measured with the log likelihood
3      q ← loglik(data, model)
4      q_old ← q + 2e-6
5      iter ← 0
6      while q - q_old ≥ 1e-6 ∧ iter < max_iter do
           // E-Step
7          foreach obs ∈ data do
8              foreach comp ∈ model do
9                  mean ← comp.mean
10                 covar ← comp.covar
11                 weight ← comp.weight
12                 prob[obs, comp] ← dnorm(obs, mean, covar) * weight
13             prob[obs, ] ← prob[obs, ] / sum(prob[obs, ])
           // M-Step
14         foreach comp ∈ model do
15             comp.mean ← weighted_mean(data, prob[, comp])
16             comp.covar ← weighted_covar(data, prob[, comp])
17             comp.weight ← sum(prob[, comp]) / rows(data)
           // Update metrics
18         q_old ← q
19         q ← loglik(data, model)
20         iter ← iter + 1
21     foreach obs ∈ data do
22         foreach comp ∈ model do
23             mean ← comp.mean
24             covar ← comp.covar
25             weight ← comp.weight
26             prob[obs, comp] ← dnorm(obs, mean, covar) * weight
27         cluster[obs] ← argmax(prob[obs, ])
28     return cluster, model
```

This algorithm was implemented in the `gaussian_mixture` function. This function is written after Algorithm 3.6. The complete source code can be found in Appendix A.3. The `gaussian_mixture` function accepts three or more arguments. The first argument, `data`, is a set of observations, presented

as a matrix-like object e.g. a `matrix` or `data.frame`, where every row is a new observation. The second argument, `k`, is an integer representing the number of clusters. The third argument, `max_iter`, is an integer representing the maximum number of iterations the EM algorithm is allowed to perform. Any additional arguments are passed to the `kmeans` function, which we previously explained in Subsection 3.2.1, whenever the `gaussian_mixture` function calls it.

Initially, the algorithm builds the `k` components of the probabilistic model. The model used is a Gaussian Mixture Model (GMM). This model is represented by Equation 3.1 where the model GMM is the sum of its components C_i and each component is a weighted Gaussian distribution parameterized by its mean μ_i, its covariance matrix Σ_i and its weight λ_i. Algorithm 3.6 initializes the model randomly. This could be done by randomly splitting the observations into `k` partitions of equal size and using the observations in partition i to compute the μ_i as the mean of the observations Σ_i as the covariance matrix of the observations and λ_i as $1/k$. However, the `gaussian_mixture` function takes a similar yet different approach where, instead of partitioning the observations randomly, the observations are partitioned with a call to *kmeans*. This is done like this since it is a computationally cheap way to obtain a good starting model.

$$\text{GMM} = \sum_{i=1}^{k} C_i = \sum_{i=1}^{k} \lambda_i \mathcal{N}(\mu_i, \Sigma_i) \tag{3.1}$$

Once the model is initialized an initial fitness metric is computed for it. The metric used by this algorithm is the log likelihood which can be computed as described in Equation 3.2. This metric will change as we optimize the model with the subsequent E and M steps. Whenever the metric changes slow down below a threshold, we can assume the model has reached (or is about to reach) a local maximum. This is therefore used as a stopping signal. In addition to this, the number of iterations of the EM algorithm is also used as a stopping signal.

$$loglik = \sum_{i=1}^{k} \sum_{obs \in data} \log\left(C_i\left(obs\right)\right) = \sum_{i=1}^{k} \sum_{obs \in data} \log\left(\lambda_i\right) + \log\left(\mathcal{N}\left(obs, \mu_i, \Sigma_i\right)\right) \tag{3.2}$$

With the model initialized and its fitness metric computed, the algorithm starts optimizing the model with iterations of the EM algorithm. During the E-Step, the probability of each observation belonging to each component is computed. These probabilities are min-maxed for every observation so that the probabilities of any specific observation belonging to each component add up to 1. During the M-Step, these probabilities are used again to compute the parameters of each component e.g. the mean of each component is computed as the weighted mean of the observations where the weight of each observation is its probability to belong to the component in question. After each M-Step, a new model is obtained and its log likelihood is computed.

When any of the stopping criteria is met, the last obtained model is the optimized version which the algorithm returns. In addition to the model, the algorithm also returns a label for each observation, indicating to which cluster they belong. To compute these labels, another E-Step is performed and, with the probabilities retrieved from the E-Step, each observation is assigned the label of the cluster to which it is more likely to belong.

3.2.4 Agglomerative Hierarchical Clustering

The Agglomerative Hierarchical Clustering (AHC) algorithm (Algorithm 3.7) performs a hierarchical cluster analysis of n observations by grouping them up into incrementally larger clusters, thus building a

hierarchy. To do this, the algorithm follows a n step process, which repeats until a single cluster remains:

1. Initially, each object is assigned to its own cluster. The matrix of distances between clusters is computed.
2. The two clusters which are closest to each other i.e. the two clusters with smaller proximity, are joined together into a new larger cluster and the proximity matrix is updated. This is done according to the specified proximity definition. This step is repeated until a single cluster containing every object remains.

Additionally, to completely understand the algorithm we also need to define how to compute the proximity between two clusters. There are several possible definitions but we are going to consider three possibilities:

1. The `single` function defines the proximity between two clusters as the distance between the closest objects among the two clusters as seen in Equation 3.3. It produces clusters where each object is closest to at least one other object in the same cluster. It is known as **SLINK**, **single-link** and **minimum-link** [129].

$$\min\{d(x, y) : x \in A, y \in B\} \tag{3.3}$$

2. `complete`. The `complete` function defines the proximity between two clusters as the distance between the furthest objects among the two clusters as seen in Equation 3.4. It is known as **CLINK**, **complete-link** and **maximum-link** [130].

$$\max\{d(x, y) : x \in A, y \in B\} \tag{3.4}$$

3. `average`. The `complete` function defines the proximity between two clusters as the average distance between every pair of objects, one from each cluster, as seen in Equation 3.5. It is known as **UPGMA** and **average-link** [131].

$$\frac{1}{|A| \cdot |B|} \sum_{x \in A} \sum_{y \in B} d(x, y) \tag{3.5}$$

This algorithm was implemented in the `agglomerative_clustering` function. This function is written after Algorithm 3.7. The complete source code can be found in Appendix A.4. This function accepts two or more arguments. The first argument, `data`, is a set of observations, presented as a matrix-like object e.g. a `matrix` or `data.frame`, where every row is a new observation. The second argument, `proximity`, is the proximity definition to be used. It has to be one of `"single"` (minimum/single linkage), `"complete"` (maximum/complete linkage) or `"average"` (average linkage). Any additional arguments are passed to the `proxy::dist` function whenever a distance is to be calculated.

Initially, the list of clusters is initialised with a cluster for each observation. Each of these clusters is a binary tree with a single observation at the root and the information of at which height the branches join i.e. what is the proximity between the clusters at each branch. In the implementation, additional information related to the amount of observations contained in the cluster and a label is stored in these trees. This information is necessary since this tree structure has to be converted to an `hclust` object to make it compatible with other R functions and to accelerate the update of the pairwise proximity matrix. Additionally, the pairwise proximity matrix is computed. Since at this point clusters are composed of a

Algorithm 3.7: Agglomerative Hierarchical Clustering

Input:
`data`. A matrix with n rows, one per observation.
`prox`. A function that computes the proximity between two clusters.
Output:
`dendro`. A binary tree structure representing the cluster hierarchy.

```
1  begin
       // Initialize the list of clusters
2      foreach obs ∈ data do
3          clusters[obs] ← {obs, height = 0}
       // Compute the pairwise proximity matrix
4      foreach o1 ∈ clusters do
5          foreach o2 ∈ clusters do
6              proximity[o1, o2] ← prox(o1, o2)
7      while size(clusters) > 1 do
           // Find which clusters to join
8          A, B ← argmin(proximity)
9          o1 ← pop(clusters, A)
10         o2 ← pop(clusters, B)
11         p1 ← pop(proximity, A, )
12         p2 ← pop(proximity, B, )
13         pop(proximity, , A)
14         pop(proximity, , B)
           // Join the clusters into a new cluster
15         o3 ← {o1, o2, height = proximity[A, B]}
16         push(clusters, o3)
17         p3 ← prox(p1, p2)
18         proximity[o3, ] ← p3
19         proximity[, o3] ← p3
20     dendro ← pop(clusters)
21     return dendro
```

single observation, this matrix is equivalent to the pairwise distance matrix of the observations. In the implementation this matrix is computed making use of the `proxy::dist` function.

With all the initial clusters in a list and until a single cluster remains in said list, two clusters are looked for such that their proximity is the smallest of all proximities in the pairwise proximity matrix. These clusters are removed from the list of clusters and combined as the branches of a new cluster. This new cluster is put back into the list of clusters and the pairwise proximity matrix is updated by removing the rows/columns of the two clusters we just joined together and replacing them with the proximity of the new cluster to every other cluster.

While the definition of proximity is often described as an operation on the observations of the clusters, the computation of the proximity can sometimes be accelerated. In our specific case, since we know the proximity between two clusters A and B to every other cluster D in the list of clusters and given the proximity definitions we are considering, the proximity between cluster $C \equiv A \cup B$ and every other cluster D can be accelerated. Let $\text{prox}(A, B)$ be the proximity between two clusters A and B, to compute $\text{prox}(C, D)$ we can combine $\text{prox}(A, D)$ and $\text{prox}(B, D)$ in some way specific to the proximity definition we are using.

Doing this for the **single-link** and **complete-link** definitions is very straight forward. In the case of **single-link**, if the proximity between to clusters is the minimum distance between any two points from those clusters, when combining two clusters A and B into C it should be obvious that the minimum distance between clusters C and D, $\text{prox}(C, D)$, is $\min(\text{prox}(A, F), \text{prox}(B, F))$. This is because the closest point in C to any point in D has to belong to either A or B therefore the minimum proximity of these two is the proximity we seek. With **complete-link** the same logic applies replacing the min function with the max function. The computation with **average-link** is a tad bit more complicated. In this case $\text{prox}(C, D)$ can be computed as:

$$\frac{|A|\,\text{prox}(A, D) + |B|\,\text{prox}(B, D)}{|A| + |B|}$$

An intuitive way to think about this is that, since $\text{prox}(A, D)$ is the average distance between the points in A and D, this could very well be the actual distance between every pair of points of these two clusters. Making this assumption we just need to compute the weighted average of $\text{prox}(A, D)$ and $\text{prox}(B, D)$ where the weights are the amount of observations in clusters A and B, $|A|$ and $|B|$ respectively.

Once a single cluster remains in the list of clusters, the algorithm simply has to return this last cluster. However, as we previously mentioned, the implementation still needs to convert this binary tree to a `hclust` object. This object represents the dendrogram by labeling every cluster and storing a two-column matrix where each row contains two of those clusters, indicating which two clusters where joined at each step. Additionally, it stores the height of each cluster merge in a separate vector too.

3.2.5 Divisive Hierarchical Clustering

The Divisive Hierarchical Clustering (DHC) algorithm (Algorithm 3.8) performs a hierarchical cluster analysis of n observations by dividing them down into smaller clusters, thus building a hierarchy. To do this, the algorithm follows a n step process, which repeats until every observation is put into a cluster of its own:

1. Initially, all objects are assigned to the same unique cluster.

2. The cluster which yields the greatest value for a criterion function e.g. the sum of squares function, is split into two sub clusters which minimize this very criterion function. This step is repeated until a every cluster contains a single observation i.e. until n clusters remain.

Algorithm 3.8: Divisive Hierarchical Clustering

Input:
`data`. A matrix with n rows, one per observation.
`criterion`. The criterion function
Output:
`dendro`. A binary tree structure representing the cluster hierarchy.

```
1  begin
       // Initialize the clusters
2      dendro ← {data}
3      push(clusters, dendro)
4      while size(clusters) > 0 do
           // Split any cluster*
5          C ← pop(clusters)
6          A, B ← split(C)
           // Update the cluster we just split as a binary tree with the two
              parts as branches
7          C ← {A, B, height = criterion(C)}
           // Add the new clusters to the list of clusters to split
8          if size(A) > 1 then
9              push(clusters, A)
10         if size(B) > 1 then
11             push(clusters, B)
12     return dendro
```

This algorithm was implemented in the `divisive_clustering` function. This function is written after Algorithm 3.8. The complete source code can be found in Appendix A.5. This function accepts one or more arguments. The first argument, `data`, is a set of observations, presented as a matrix-like object e.g. a `matrix` or `data.frame`, where every row is a new observation. Any additional arguments are passed to the `kmeans` function, which we previously explained in Subsection 3.2.1, whenever the function splits a cluster into two.

Initially, a global cluster is initialized as a tree with all the data in the root and no branches. In the implementation, some extra information is added to the clusters. This information comprises a label identifying the cluster, the amount of elements contained in the cluster and the value the cluster yields from the criterion function. In this case, the implementation uses the sum of squares criterion function. The function needs this extra information to convert the tree structure from Algorithm 3.8 into a `hclust` object. Additionally, a list of clusters which have to be split is created and the global cluster is added to it.

Once the list of clusters to split is initialized, any cluster is taken out of the list and said cluster is processed. Despite the fact that the algorithm states that the cluster which maximizes the criterion function is the one which should be split, taking any cluster from the list of clusters on the leaf nodes of the global cluster, as shown in Line 4 of Algorithm 3.8, is fine. This is due to two reasons. The first reason is that we are fully expanding the global cluster until leafs contain a single observation, instead of stopping early once we hit a certain amount of clusters. The second reason is that once a cluster is split into two, each of the two new clusters will be expanded into a tree completely independent from that of the other cluster. Therefore, no matter the order in which the clusters are expanded, the end result will

be the same.

Once we have selected a cluster to work with, we have to determine how to split it. In Algorithm 3.8 this is done with the `split` sub-function which could exhaustively check every possible bi-partition of the cluster in search of the one which minimizes the criterion function. However, since that would be computationally prohibitive, this implementation splits the clusters into two using the `kmeans` function. This is done like so because the K-Means algorithm is very fast, and allows us to choose the amount of partitions we want. That is also the reason why the criterion function used is the sum of squares, since that is the function the K-Means algorithm optimizes. Therefore, this implementation of the DHC resembles a Bisecting K-Means algorithm. Nevertheless, this approach is not without its problems. One of the possible problems is that the K-Means algorithm may sometimes yield a single cluster instead of the two we need, meaning we will need to tune the algorithm to adapt to this situations. Another problem with this approach is that the implementation will inherit some of the shortcomings of the K-Means algorithm. Once the cluster has been split into two, the old cluster is updated to reflect this division.

In addition, the split clusters are added to the list of clusters which have to be divided, if they contain more than a single observation and can therefore be split. When no clusters remain in the list of clusters to split, the algorithm returns the global cluster. In the case of the R implementation, this tree has to be converted to a `hclust` object. In this case, since the splits have not been performed in decreasing order of values yielded by the criterion function, the order of the cluster "merges" also has to be sorted. This makes this conversion process more convoluted than the corresponding process of the `agglomerative_clustering` function.

3.3 Automatic explanations

While we have now explained in some detail how each of these algorithms work and how the implementations differ from the algorithms, a student using this package will (most likely) not be able to read these explanations. Instead, for each implemented method the package provides the option to log to the console details about the steps the method in question is taking to cluster the input data. These explanations are articulated through two additional parameters: `details`, which determines whether or not these details should be logged or not; and `waiting`, which determines whether or not user input should be waited for before continuing with the next step of the algorithm.

In general, the explanations follow the same steps for each of the implementations. Initially, the algorithm is explained in natural language, usually giving a list of steps, so that the student knows what to expect of the coming logs. Once the algorithm has been explained, the actual execution starts. During the execution, after any of the steps given in the natural language explanation happens, the relevant information about the step is logged along with some details of what is being logged. Finally, once the algorithm terminates, some extra logs explaining the results are given. After each of these logs, the execution may (depending on the value of the `waiting` parameter) halt and wait for some user input before continuing.

In order to make the logs as readable as possible, a color code is used. With the default RStudio theme, the explanations and details of each step are logged in red, the relevant information and specific data about certain steps is logged in black, and the prompts to wait for user input are logged in blue. To get this result, different functions are used, some of which can be found in Appendix A.6. On one hand, the custom function `console.log` uses the native R function `message` which prints in red. On the other hand, both `print` and `cat` properly print matrices and vectors in black color. Finally, the `readline` function can print a prompt in blue. And of course, these colors change with the IDE theme.

These color codes help make the explanation easier to follow, since it gives a visual key as to what the user is looking at.

Chapter 4

Results

In this chapter we will have a look at the pros and cons of the implemented clustering algorithms and compare them by clustering six toy datasets included in the package and looking a the results each algorithm yields. Additionally, the automatic explanations provided by the implementations will be showcased with an example.

4.1 Toy Datasets

The techniques explored and implemented in this project are clustering algorithms. As such, they always take some data as input, and return some classification of the data as output. Therefore, to put the implementations to the test, we will need some input data.

A toy dataset is a small dataset meant for testing purposes. The R programming language ships with some very famous toy datasets like Edgar Anderson's Iris Data [163], a dataset with some measurements for 50 flowers from each of 3 species of iris: setosa, versicolor and virginica. While these datasets are alright, they are not meant for clustering specifically and may require some treatment of the data before hand.

Instead of relying on R's toy datasets, this project implements its own datasets based on scikit-learn's [164] datasets with noise from its clustering methods overview. These datasets are considered better for several reasons. On one side, they are meant specifically for clustering e.g. they don't have predefined classes like R's iris does. On the other side, they are designed to highlight different features of the clustering algorithms, allowing the user to better understand them. Additionally, although all of the implementations from this project are designed to work on n dimensional data, these datasets are bidimensional which is easier to plot and thus understand. Figure 4.1 shows these toy datasets.

4.2 Algorithm Characteristics

In this section we will discuss each of the algorithm's characteristics, both positive and negative, and use both the implementations and the toy datasets to visually see some of these characteristics in action. Before diving into the characteristics of any specific algorithm, a brief summary of how the algorithm work will be given. This is due to the fact that some of their flaws and qualities are directly derived from from some specific mechanism used by the algorithm in question.

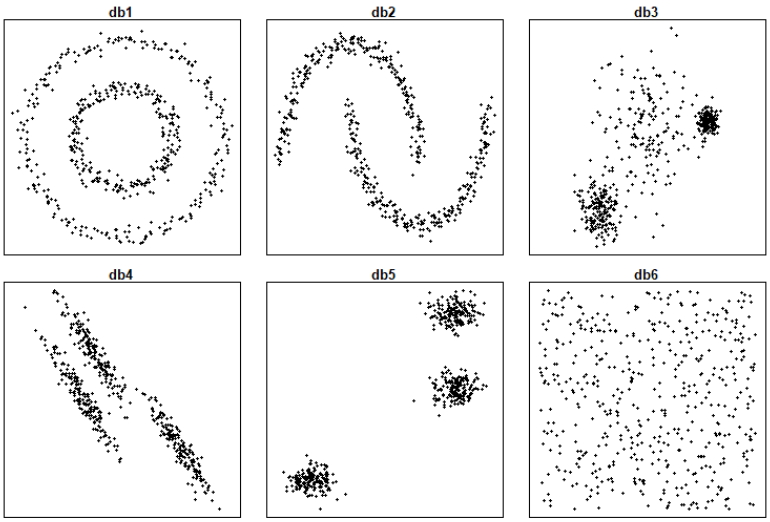


Figure 4.1: Toy Datasets

4.2.1 K-Means

The K-Means algorithm clusters data by trying to separate samples in k groups such that they minimize some criterion function, specifically, the within-cluster sum-of-squares. The within-cluster sum-of-squares can be recognized as a measure of how internally coherent clusters are. Given this criterion function, a cluster can be represented by the mean of the observations in that cluster i.e. the cluster centroid. On one hand, among the virtues of this algorithm we can find the following:

- This algorithm is relatively simple to understand and implement.
- Since it converges quickly, it can handle a large number of observations efficiently.
- When clusters are well-separated and of similar size, it tends to perform well.

On the other hand, it algorithm also suffers from various drawbacks:

- The target number of clusters needs to be provided in advance which may not always be clear, specially in real-world scenarios (Figure 4.2).
- The initial placement of the centroids can negatively affect the convergence, leading to sub-optimal solutions.
- If there are outliers present in the data, they can heavily influence the centroid placement.
- Due to the fact that this algorithm optimizes the within-cluster sum-of-squares, the clusters are assumed to be convex, isotropic and of similar size and density. If clusters are of different shapes, size or density; the algorithm might struggle (Figure 4.2).
- Once again, due to the fact that this algorithm optimizes the within-cluster sum-of-squares, the topology of the data is not considered. Therefore, clusters with complex shapes may not be captured (Figure 4.2).
- When clustering very high-dimensional data, the Euclidean distances tend to be inflated. This is called the “curse of dimensionality”. The “curse of dimensionality” hinders the performance of the algorithm.

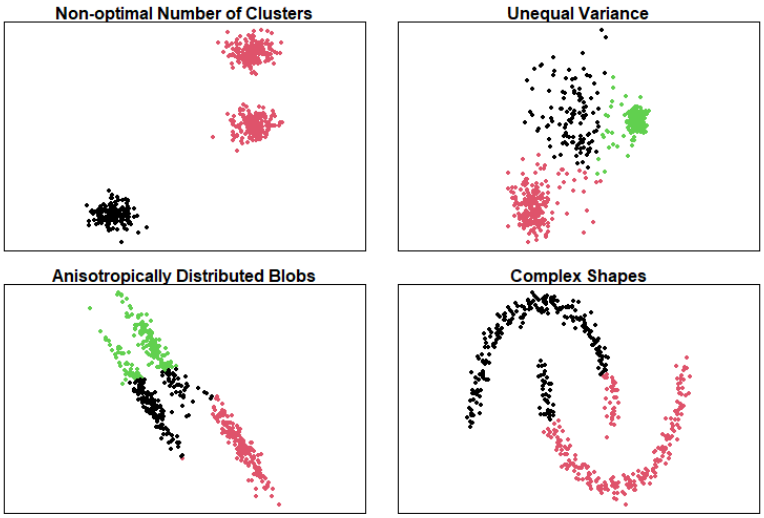


Figure 4.2: K-Means - Drawbacks

4.2.2 DBSCAN

The DBSCAN algorithm views clusters as areas of high density separated by areas of low density. Due to this rather generic view, clusters found by DBSCAN can be any shape, as opposed to K-Means which assumes that clusters are convex shaped. The central component to the DBSCAN is the concept of core samples, which are samples that are in areas of high density. A cluster is therefore a set of core samples, each close to each other and a set of non-core samples that are close to a core sample. This algorithm has many features which make it attractive to use:

- It can identify clusters of arbitrary shapes (Figure 4.3).
- Unlike K-Means, DBSCAN does not require to know the number of clusters beforehand. It can automatically determine the number of clusters based on the data and its density characteristics (Figure 4.3).
- It is robust to noise and outliers, labeling them as noise instead of assigning them to a cluster (Figure 4.3).

However, it too has its drawbacks:

- Unless it makes use of data structures like kd-trees or B*-trees, it can be time and memory intensive hindering its scaling capabilities.
- It may struggle when dealing with clusters of varying densities (Figure 4.3).
- It might not perform properly when clusters are close to each other.
- Like other clustering algorithms, it can face challenges in high-dimensional spaces due to the “curse of dimensionality”.

4.2.3 Gaussian Mixture

The Gaussian Mixture method clusters data by trying to model the data samples with a Gaussian Mixture Model. This model is optimized for log-likelihood using the Expectation Maximization algorithm. The

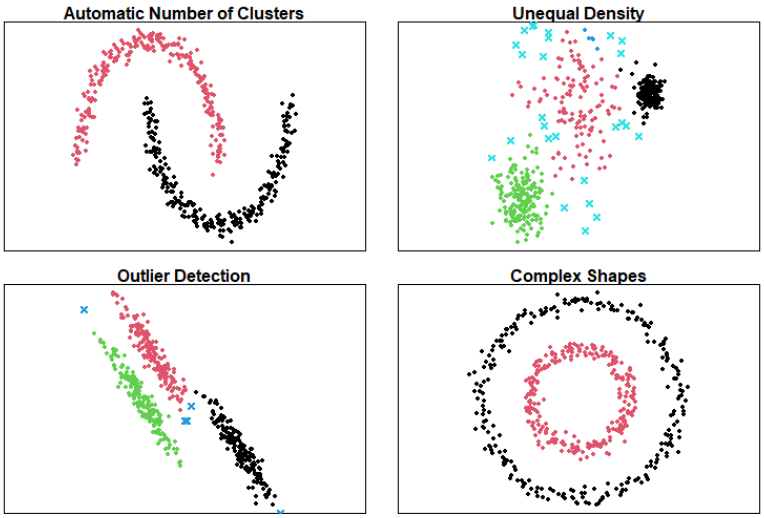


Figure 4.3: DBSCAN - Pros & Cons

log likelihood can be interpreted as a measure of how likely is for a probability distribution to produce a set of observations. This algorithm, represents clusters as observations from a Gaussian Model. This way to cluster data brings along many benefits:

- A GMM can model clusters with shapes more flexible than those of K-Means. They do not need to be isotropic, but can now be elongated (Figure 4.4).
- Cluster assignments are soft i.e. observations are assigned a probability to belong to each cluster. This can be more realistic than the hard assignment of K-Means in some scenarios.
- Clusters are not assumed to have similar sizes or densities, making it suitable for datasets with such characteristics (Figure 4.4).

However, as with every other clustering algorithm, GMM with EM has some problems:

- Just like with K-Means, GMM with EM is sensitive to the initial value of the model. Poor initialization can lead to sub-optimal solutions (Figure 4.4).
- This algorithm involves iterative optimization, which can be computationally intensive and time-consuming. For this reason, it is almost impossible to scale.
- It can struggle with high-dimensional data due to the increased complexity of estimating multivariate Gaussian distributions accurately.
- The clusters modeled by GMM with EM need to follow a Gaussian distribution. Therefore, it is not as flexible as DBSCAN when it comes to cluster shapes (Figure 4.4).

4.2.4 Agglomerative Hierarchical Clustering

The AHC algorithm builds a cluster hierarchy by creating a cluster for each observation and merging clusters together into increasingly bigger clusters until a single cluster remains. This hierarchy is represented as a tree (or dendrogram) where the root of the tree is the cluster containing all samples and the leaves are the clusters containing a single sample. On the positive side, this algorithm has the following characteristics:

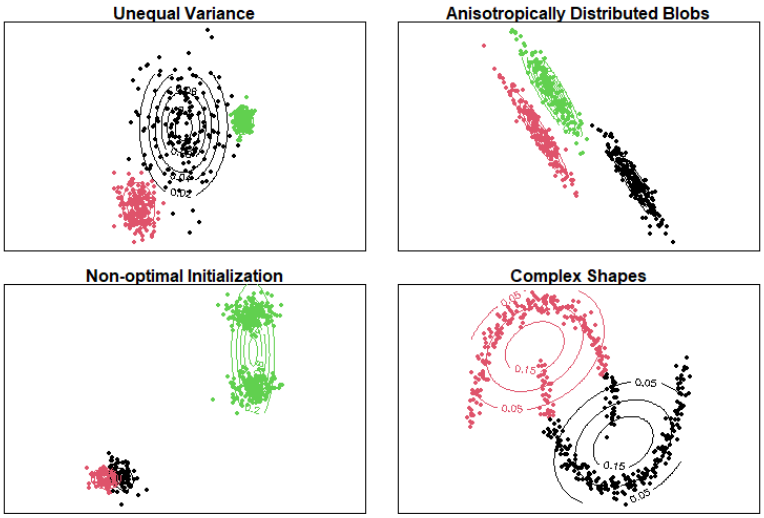


Figure 4.4: GMM with EM - Characteristics

- It produces a hierarchy which can be used to gain insight onto the structure of the data at different levels of granularity.
- It does not require the user to specify the number of clusters in advance. Instead, the hierarchy can be explored to determine the number of clusters that best fits the data.
- It is not sensitive to initialization as it always produces the same outcome for the same inputs (unlike K-Means).
- Different distance metrics can be used to measure the similarity between clusters or data points, allowing the it to adapt to different types of data (Figure 4.5).

Meanwhile, on the negative side, this algorithm has the following hurdles to overcome:

- It can be computationally intensive, as the time complexity is at best quadratic (and the memory complexity is not good either). Therefore, it does not scale well.
- It is sensitive to noise and outliers, potentially leading to erroneous merges (Figure 4.5).
- Once a merge is done, it cannot be undone. This can lead to sub-optimal results if an early merging decision is incorrect.
- Determining the optimal number of clusters from the hierarchy can be subjective and depends on the specific problem (Figure 4.5).

4.2.5 Divisive Hierarchical Clustering

The DHC algorithm builds a cluster hierarchy by creating a cluster which contains every single observation and dividing it into sub-clusters until the clusters contain a single observation. This hierarchy is represented as a tree (or dendrogram) where the root of the tree is the cluster containing all samples and the leaves are the clusters containing a single sample. On the positive side, this algorithm has the following characteristics:

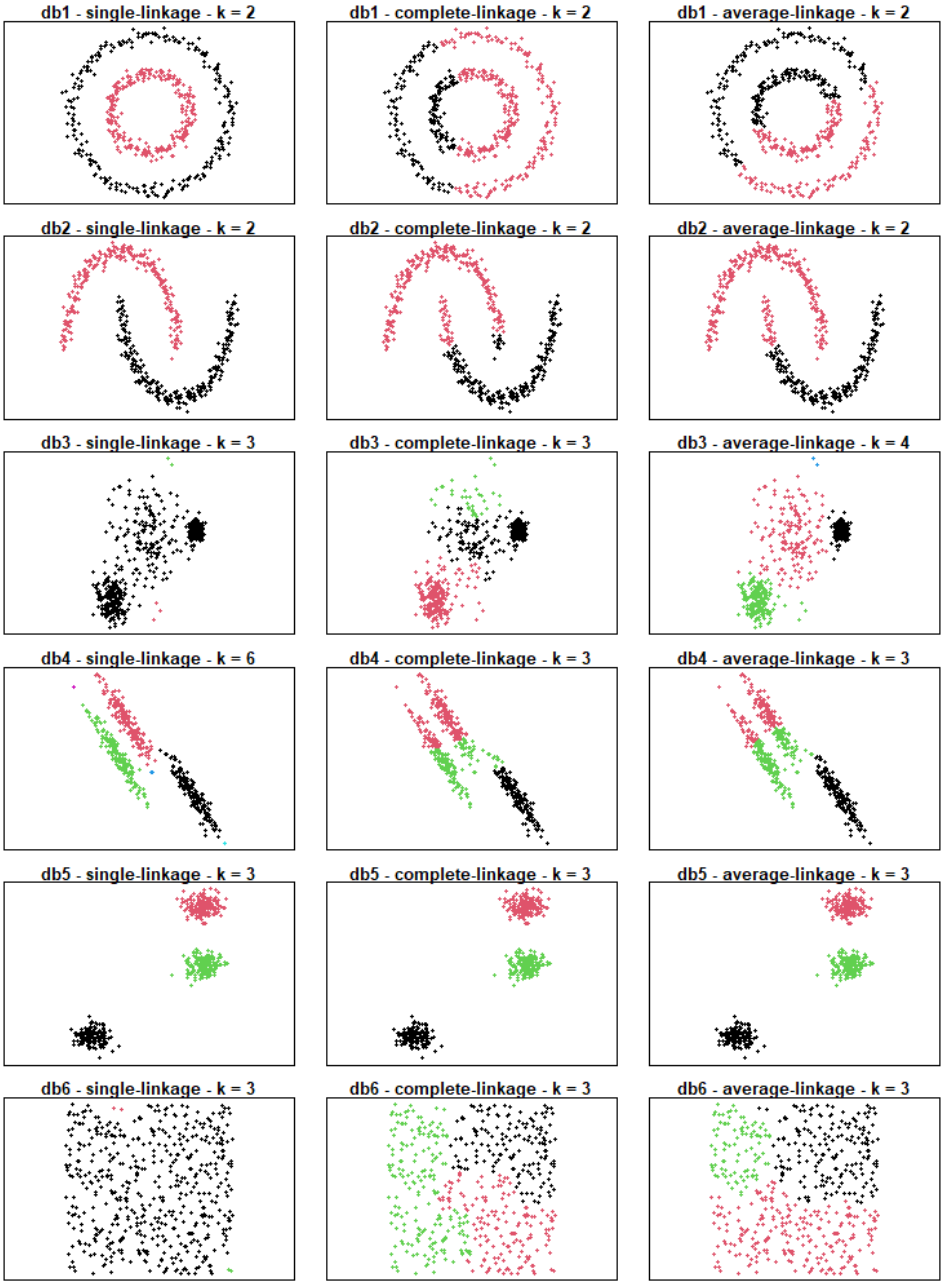


Figure 4.5: Agglomerative Hierarchical Clustering - Linkage strategies

- It produces a hierarchy which can be used to gain insight onto the structure of the data at different levels of granularity.
- It does not require the user to specify the number of clusters in advance. Instead, the hierarchy can be explored to determine the number of clusters that best fits the data.
- It starts with a single cluster and divides it into smaller clusters based on the internal structure of the data, which potentially leads to more meaningful and natural cluster divisions.
- Different distance metrics can be used to measure the similarity between clusters or data points, allowing the it to adapt to different types of data.

Meanwhile, on the negative side, this algorithm has the following hurdles to overcome:

- It can be computationally expensive, specially for large datasets, as it repeatedly divides the data, optimizing a metric, until individual data points become clusters. Therefore, it does not scale well.
- It is sensitive to noise and outliers, potentially leading to sub-optimal divisions.
- Once a division is done, it cannot be undone. This can lead to sub-optimal results if an early dividing decision is incorrect.
- Determining the optimal number of clusters from the hierarchy can be subjective and depends on the specific problem.
- It can be more complex to implement than agglomerative clustering algorithms as it needs recursive division and handling of the results.

4.3 Algorithm comparison

Having seen the individual characteristics of each of the algorithm, this section will compare every algorithm side by side on all toy datasets. In order for the comparisons to be fair, the best configuration of each algorithm is chosen for each dataset. Additionally, the time it took for every algorithm to finish clustering each dataset was measured. All of these results are displayed in Figure 4.6.

The first dataset, `db1`, contains data resembling two concentric circumferences where each circumference corresponds to a cluster. The second dataset, `db2`, contains data resembling two semicircles which are slightly misaligned. In both cases, the observations are subject to some noise. In these scenarios, given how the clusters are positioned with respect to each other, we could consider them to have complicated shapes. For this reason, K-Means, GMM and DHC perform poorly drawing a straight line through the space an assigning the observations on each side to a different cluster. AHC also has a hard time solving these scenarios, with complete-linkage and average-linkage producing similarly bad results. However, both DBSCAN and AHC with single-linkage connect observations which are nearby, successfully clustering the complex shapes of `db1` and `db2`. In general, in any situation where the ability of DBSCAN to connect nearby observations is needed to find the clusters, AHC's single-linkage will also produce satisfactory results. Nevertheless, AHC takes a much longer time to do this as it does not only divide the observations in a few groups but it builds a complete hierarchy.

The third dataset, `db3`, contains data generated by three different Gaussian distributions. As such, this scenario is perfect for the GMM algorithm as it clusters using exactly the same probabilistic model. In this scenario DBSCAN is the worst performer as it classifies many observations as outliers because it does

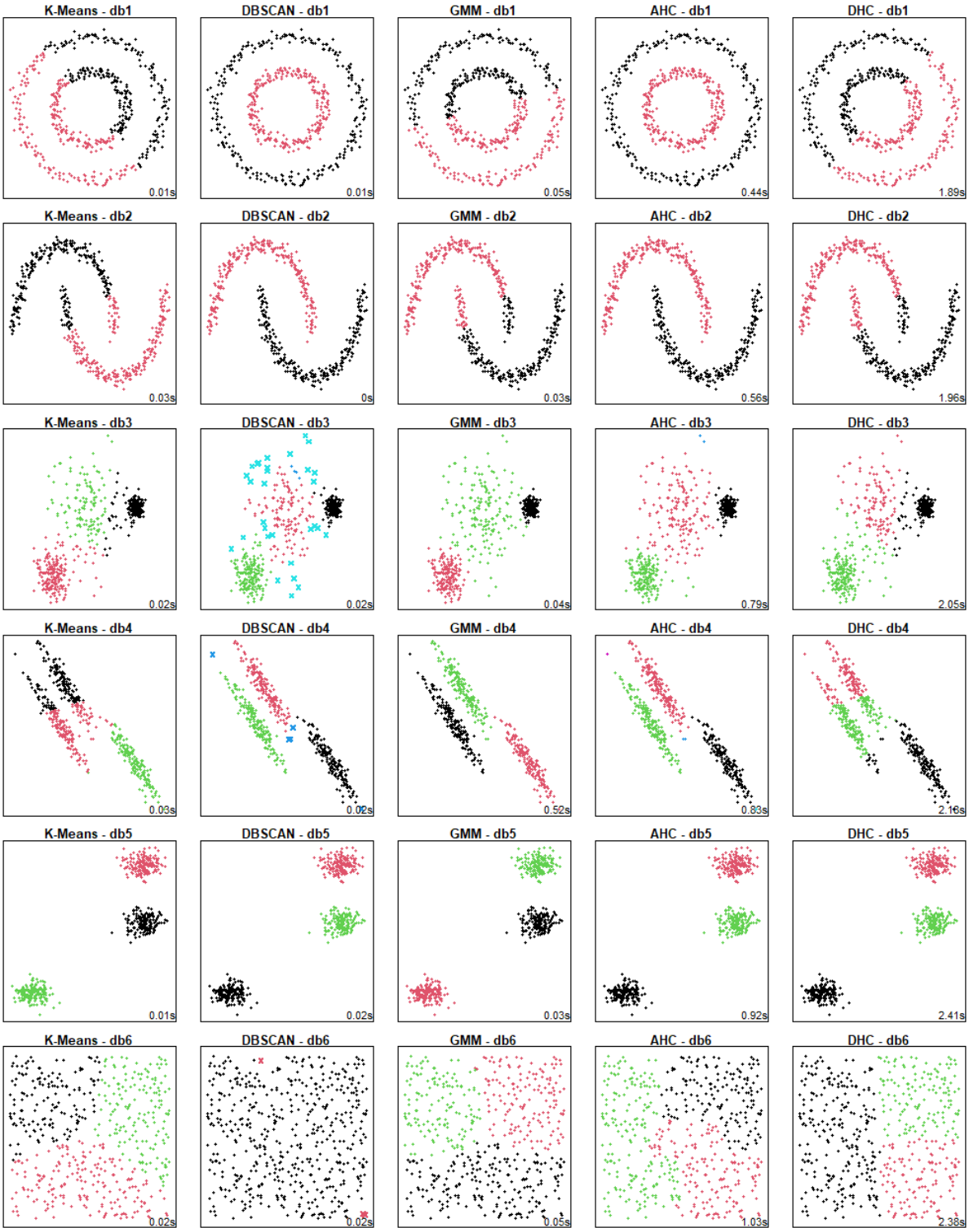


Figure 4.6: Clusterization of every toy dataset with every algorithm

not adapt well to clusters with varying densities. K-Means and DHC follow shortly behind misclassifying many observations due to the fact that clusters are of different size. Lastly, AHC with average-linkage gets results comparable to those of GMM. To achieve this results with AHC we need to split the data into four clusters. Doing this, one of the clusters contains only two observations which we could interpret as outliers.

The fourth dataset, `db4`, contains data resembling three diagonally elongated clusters. In this case, both K-Means and DHC fail to capture the clusters correctly, completely mixing two of the clusters together. Once again, this is due to the criterion function these algorithms use, which expects clusters to be convex, isotropic and of similar size and density. DBSCAN and AHC with single-linkage manage to capture the clusters since, unlike in `db3`, they are properly separated. In the case of AHC we need to split the data in six clusters, where three of the clusters are very small (containing up to two observations). We can consider these small clusters outliers. Additionally, the GMM also manages to correctly identify the clusters since they can indeed be represented by Gaussian distributions. However, GMM does a better job than DBSCAN and AHC since it classifies no observations as outliers.

The fifth dataset, `db5`, contains data resembling three clusters which are convex, isotropic well separated and of similar size and density. This time, all algorithms manage to get satisfactory results.

Lastly, the sixth dataset, `db6`, contains uniformly distributed data forming no obvious clusters. This scenario is interesting since it somehow allows us to see how the algorithms work internally. On the one hand, we can see how both K-Means and GMM form all clusters simultaneously since they split the plane in three similar parts. On the other hand, this scenario makes obvious that DHC splits clusters into two every time, dividing the data into halves and the halves into fourths. Lastly, it makes obvious how DBSCAN and AHC with single-linkage work in a somewhat similar way, and how varied the results obtained with each linkage method can be.

To wrap up, it is also interesting to see how the hierarchical algorithms always take much longer than their partitional counterparts. As we mentioned before, this is due to the fact that partitional algorithms only produce a set amount of partitions which is much smaller than the number of observations. However, hierarchical algorithms have to produce a complete hierarchy i.e. partition the dataset as many times as there are observations. This makes hierarchical algorithms much more computationally complex than partitional ones.

4.4 Automatic explanations

To exemplify the automatic explanations we will see the steps they tend to follow with the details of an AHC. To do this, we will once again resort to one of the toy datasets, in this case `db5`. However, instead of using the complete dataset, which contains 500 observations, we will only use a very small fraction of the data, consisting of the first six observations. This is to make sure the logs will be of data we can fit in the width of a terminal.

Initially, the algorithm is explained in natural language, usually giving a list of steps, so that the student knows what to expect of the coming logs. We can see this in Figure 4.7 where an explanation very similar to that of the documentation is given.

Once the algorithm has been explained, the actual execution starts. During the execution, after any of the steps given in the natural language explanation happens, the relevant information about the step is logged along with some details of what is being logged. This is showcased in Figures 4.8 - 4.9, where both the initial step and the second (nth) step are explained in a fashion similar to that of the explanation given beforehand.

```
> cl <- clustlearn::agglomerative_clustering(
+     clustlearn::db5[1:6, ],
+     'single',
+     details = TRUE,
+     waiting = TRUE
+ )
--------------------------------------------------------------------------------
EXPLANATION:

The Agglomerative Hierarchical Clustering algorithm defines a clustering hierarchy for
 a dataset following a `n` step process, which repeats until a single cluster remains:

    1. Initially, each object is assigned to its own cluster. The matrix of distances
    between clusters is computed.
    2. The two clusters with closest proximity will be joined together and the proximi
    ty matrix updated. This is done according to the specified proximity. This step is
     repeated until a single cluster remains.

The definitions of proximity considered by this function are:

    1. `single`. Defines the proximity between two clusters as the distance between th
    e closest objects among the two clusters. It produces clusters where each object i
    s closest to at least one other object in the same cluster. It is known as SLINK,
    single-link or minimum-link.
    2. `complete`. Defines the proximity between two clusters as the distance between
    the furthest objects among the two clusters. It is known as CLINK, complete-link o
    r maximum-link.
    3. `average`. Defines the proximity between two clusters as the average distance b
    etween every pair of objects, one from each cluster. It is also known as UPGMA or
    average-link.

Press [enter] to continue
```

Figure 4.7: Automatic explanation for AHC (Part 1)

```
--------------------------------------------------------------------------------
STEP 1:

Initially, each object is assigned to its own cluster. This leaves us with the followi
ng clusters:
CLUSTER #-1 (size: 1)
          x         y
1 -1.578117 -1.292868
CLUSTER #-2 (size: 1)
          x        y
2 0.7027994 1.193823
CLUSTER #-3 (size: 1)
          x        y
3 0.7854535 1.191428
CLUSTER #-4 (size: 1)
          x           y
4 0.6757613 -0.04002442
CLUSTER #-5 (size: 1)
          x         y
5 0.8484305 0.2230609
CLUSTER #-6 (size: 1)
          x        y
6 0.515489 0.3014147

Press [enter] to continue

The matrix of distances between clusters is computed:
Distances:
      -1    -2    -3    -4    -5
-2 3.374
-3 3.429 0.083
-4 2.579 1.234 1.236
-5 2.861 0.982 0.970 0.315
-6 2.632 0.912 0.930 0.377 0.342

Press [enter] to continue
```

Figure 4.8: Automatic explanation for AHC (Part 2)

```
STEP 2:

The two clusters with closest proximity are identified:
Clusters:
CLUSTER #-2 (size: 1)
CLUSTER #-3 (size: 1)
Proximity:
[1] 0.08268877

Press [enter] to continue

They are merged into a new cluster:
CLUSTER #1 (size: 2) [CLUSTER #-2 + CLUSTER #-3]

Press [enter] to continue

The proximity matrix is updated. To do so the rows/columns of the merged clusters are
removed, and the rows/columns of the new cluster are added:
Distances:
      -1    -4    -5    -6
-4 2.579
-5 2.861 0.315
-6 2.632 0.377 0.342
1  3.374 1.234 0.970 0.912

Press [enter] to continue
```

Figure 4.9: Automatic explanation for AHC (Part 3)

Finally, once the algorithm terminates, some extra logs explaining the results are given. Figure 4.10 shows the last step where the matrix of pairwise proximities is empty, indicating the algorithm has come to an end. Right after that, the results for the execution are shown, in this case by plotting the dendrogram of the hierarchy.

```
STEP 6:

The two clusters with closest proximity are identified:
Clusters:
CLUSTER #-1 (size: 1)
CLUSTER #4 (size: 5)
Proximity:
[1] 2.578678

Press [enter] to continue

They are merged into a new cluster:
CLUSTER #5 (size: 6) [CLUSTER #-1 + CLUSTER #4]

Press [enter] to continue

The proximity matrix is updated. To do so the rows/columns of the merged clusters are
removed, and the rows/columns of the new cluster are added:
Distances:
dist(0)

Press [enter] to continue

RESULTS:

Since all clusters have been merged together, the final clustering hierarchy is:
(Check the plot for the dendrogram representation of the hierarchy)

Press [enter] to continue
```

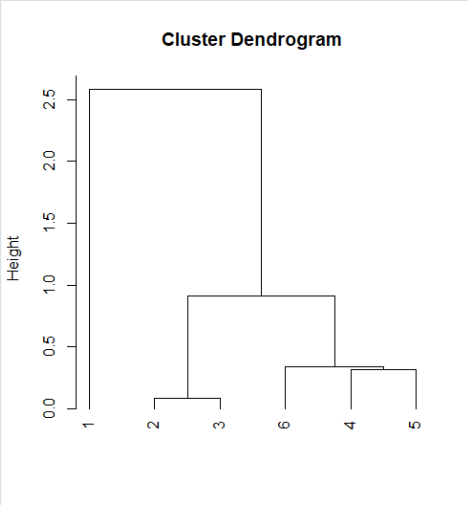


Figure 4.10: Automatic explanation for AHC (Part 4)

4.5 Uploading a package to CRAN

Having checked that the package functions correctly i.e. the algorithms perform within our expectations and the explanations are satisfactory, the next step in its life-cycle is to upload it to a public repository for teachers and students to use. The Comprehensive R Archive Network (CRAN) is a network of FTP and web servers around the world that store identical, up-to-date, versions of code and documentation for R. The CRAN is maintained by the maintainers of the R language, making it the the official repository of R packages.

To upload a package to CRAN, the devtools package provides the function `release()`. This function

asks the developer whether or not they have performed a number of different tests on the package, to make sure it is up to CRAN's standards. Once the developer confirms the package passed all tests, it is sent to the CRAN for revision. The package is tested automatically and in some instances (like the package being new to CRAN) manually. If all revisions are successful the package is finally uploaded and can be easily installed from any computer with R and an internet connection with the function `install.packages()`.

In the specific case of this project, the developed package goes by the name clustlearn. Once all desired functionality was implemented, some necessary files where created (README.md, which serves as an introduction to the package and specifies how to install it; NEWS.md, which provides a list of what features where introduced in each version of the package; and CRAN-COMMENTS, which is a way for developers to communicate with CRAN's revision team) and the package was uploaded with the previously explained `release()` function. As a result, the results of this project can be found in CRAN and installed from it.

Chapter 5

Conclusions and Future Work

To wrap up, the objective of developing a package to ease the process of learning about clustering techniques and have students get familiar with them can be considered fulfilled. Of the ten distinct categories of clustering algorithms presented in Chapter 2, at least six can be considered to be covered by the developed package. The result of all this work is the package `clustlearn`.

`clustlearn` improves on other packages which aimed to solve the very same problem like `LearnClust`. In this specific case, it does so by fixing several problems `LearnClust` had like: wrong input data handling, which sometimes led to exceptions; fixed dimensionality, since it could only handle two-dimensional data; slow run-times, which made clustering datasets like `clustlearn`'s toy datasets unfeasible; and messy explanations, sometimes relying on images the package did not ship.

While `clustlearn` is a fully functional tool, it could do with some improvements. For instance, more algorithms could be implemented to provide a broader representation of this field of study, closer to that of the state-of-the-art (for instance, no automatic clustering functions are included in `clustlearn`). Additionally, many of `clustlearn`'s implementations are not optimized. This could be a problem since it limits the maximum size a dataset can have before clustering it becomes problematic. While that does not interfere with the explanations it does limit the student's capacity to play around with the algorithms and hinders the process of getting familiar with them.

Bibliography

[1] U. Von Luxburg, R. C. Williamson, and I. Guyon, "Clustering: Science or art?" In *Proceedings of ICML workshop on unsupervised and transfer learning*, JMLR Workshop and Conference Proceedings, 2012, pp. 65–79.

[2] A. Ghosal, A. Nandy, A. K. Das, S. Goswami, and M. Panday, "A short review on different clustering techniques and their applications," *Emerging Technology in Modelling and Graphics: Proceedings of IEM Graph 2018*, pp. 69–83, 2020.

[3] D. Dua and C. Graff, *UCI machine learning repository*, 2017. [Online]. Available: http://archive.ics.uci.edu/ml.

[4] D. Chen, *Online Retail II*, UCI Machine Learning Repository, DOI: 10.24432/C5CG6D, 2019.

[5] H. Meek Thiesson, *US Census Data (1990)*, UCI Machine Learning Repository, DOI: 10.24432/C5VP42.

[6] *Estimation of obesity levels based on eating habits and physical condition*, UCI Machine Learning Repository, 2019.

[7] D. L. Haury and P. Rillero, "Perspectives of hands-on science teaching.," 1994.

[8] A. T. Lumpe and J. S. Oliver, "Dimensions of hands-on science," *The American Biology Teacher*, pp. 345–348, 1991.

[9] D. A. Bergin, "Influences on classroom interest," *Educational psychologist*, vol. 34, no. 2, pp. 87–98, 1999.

[10] A. Krapp, "Basic needs and the development of interest and intrinsic motivational orientations," *Learning and instruction*, vol. 15, no. 5, pp. 381–395, 2005.

[11] U. Schiefele, "Interest, learning, and motivation," *Educational psychologist*, vol. 26, no. 3-4, pp. 299–323, 1991.

[12] S. Hidi and K. A. Renninger, "The four-phase model of interest development," *Educational psychologist*, vol. 41, no. 2, pp. 111–127, 2006.

[13] J. W. Renner *et al.*, "Secondary school students' beliefs about the physics laboratory.," *Science Education*, vol. 69, no. 5, pp. 649–63, 1985.

[14] S. Hebbar, P. Pattar, and V. Golla, "A mobile zigbee module in a traffic control system," *IEEE Potentials*, vol. 35, no. 1, pp. 19–23, 2016.

[15] J. T. Burman, C. D. Green, and S. Shanker, "On the meanings of self-regulation: Digital humanities in service of conceptual clarity," *Child development*, vol. 86, no. 5, pp. 1507–1521, 2015.

[16] D. L. Butler and P. H. Winne, "Feedback and self-regulated learning: A theoretical synthesis," *Review of educational research*, vol. 65, no. 3, pp. 245–281, 1995.

[17] P. H. Winne and N. E. Perry, "Measuring self-regulated learning," in *Handbook of self-regulation*, Elsevier, 2000, pp. 531–566.

[18] N. E. Perry, L. Phillips, and L. Hutchinson, "Mentoring student teachers to support self-regulated learning," *The elementary school journal*, vol. 106, no. 3, pp. 237–254, 2006.

[19] B. J. Zimmerman, "Self-regulated learning and academic achievement: An overview," *Educational psychologist*, vol. 25, no. 1, pp. 3–17, 1990.

[20] M. Boekaerts and L. Corno, "Self-regulation in the classroom: A perspective on assessment and intervention," *Applied psychology*, vol. 54, no. 2, pp. 199–231, 2005.

[21] S. G. Paris and A. H. Paris, "Classroom applications of research on self-regulated learning," *Educational psychologist*, vol. 36, no. 2, pp. 89–101, 2001.

[22] C. S. Dweck and E. L. Leggett, "A social-cognitive approach to motivation and personality.," *Psychological review*, vol. 95, no. 2, p. 256, 1988.

[23] Y. Zhou, H. Wu, Q. Luo, and M. Abdel-Baset, "Automatic data clustering using nature-inspired symbiotic organism search algorithm," *Knowledge-Based Systems*, vol. 163, pp. 546–557, 2019.

[24] L. M. Abualigah, A. T. Khader, and E. S. Hanandeh, "A new feature selection method to improve the document clustering using particle swarm optimization algorithm," *Journal of Computational Science*, vol. 25, pp. 456–466, 2018.

[25] D.-X. Chang, X.-D. Zhang, C.-W. Zheng, and D.-M. Zhang, "A robust dynamic niching genetic algorithm with niche migration for automatic clustering problem," *Pattern recognition*, vol. 43, no. 4, pp. 1346–1360, 2010.

[26] Y. Liu, X. Wu, and Y. Shen, "Automatic clustering using genetic algorithms," *Applied mathematics and computation*, vol. 218, no. 4, pp. 1267–1279, 2011.

[27] A. E. Ezugwu, A. M. Ikotun, O. O. Oyelade, *et al.*, "A comprehensive survey of clustering algorithms: State-of-the-art machine learning applications, taxonomy, challenges, and future research prospects," *Engineering Applications of Artificial Intelligence*, vol. 110, p. 104 743, 2022, ISSN: 0952-1976. DOI: https://doi.org/10.1016/j.engappai.2022.104743. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S095219762200046X.

[28] R. Xu and D. Wunsch, "Survey of clustering algorithms," *IEEE Transactions on neural networks*, vol. 16, no. 3, pp. 645–678, 2005.

[29] S. Singh and S. Srivastava, "Review of clustering techniques in control system: Review of clustering techniques in control system," *Procedia Computer Science*, vol. 173, pp. 272–280, 2020.

[30] K. Djouzi and K. Beghdad-Bey, "A review of clustering algorithms for big data," in *2019 International Conference on Networking and Advanced Systems (ICNAS)*, IEEE, 2019, pp. 1–6.

[31] A. E. Ezugwu, "Nature-inspired metaheuristic techniques for automatic clustering: A survey and performance study," *SN Applied Sciences*, vol. 2, pp. 1–57, 2020.

[32] D. Xu and Y. Tian, "A comprehensive survey of clustering algorithms," *Annals of Data Science*, vol. 2, pp. 165–193, 2015.

[33] A. C. Benabdellah, A. Benghabrit, and I. Bouhaddou, "A survey of clustering algorithms for an industrial context," *Procedia computer science*, vol. 148, pp. 291–302, 2019.

[34] A. Fahad, N. Alshatri, Z. Tari, *et al.*, "A survey of clustering algorithms for big data: Taxonomy and empirical analysis," *IEEE transactions on emerging topics in computing*, vol. 2, no. 3, pp. 267–279, 2014.

[35] Z. Dafir, Y. Lamari, and S. C. Slaoui, "A survey on parallel clustering algorithms for big data," *Artificial Intelligence Review*, vol. 54, pp. 2411–2443, 2021.

[36] A. Saxena, M. Prasad, A. Gupta, *et al.*, "A review of clustering techniques and developments," *Neurocomputing*, vol. 267, pp. 664–681, 2017.

[37] A. Nagpal, A. Jatain, and D. Gaur, "Review based on data clustering algorithms," in *2013 IEEE conference on information & communication technologies*, IEEE, 2013, pp. 298–303.

[38] J. Oyelade, I. Isewon, F. Oladipupo, *et al.*, "Clustering algorithms: Their application to gene expression data," *Bioinformatics and Biology insights*, vol. 10, BBI–S38316, 2016.

[39] K. Bindra and A. Mishra, "A detailed study of clustering algorithms," in *2017 6th international conference on reliability, infocom technologies and optimization (trends and future directions)(ICRITO)*, IEEE, 2017, pp. 371–376.

[40] A. K. Jain and R. C. Dubes, *Algorithms for clustering data*. Prentice-Hall, Inc., 1988.

[41] A. K. Jain, "Data clustering: 50 years beyond k-means," *Pattern recognition letters*, vol. 31, no. 8, pp. 651–666, 2010.

[42] A. K. Jain, M. N. Murty, and P. J. Flynn, "Data clustering: A review," *ACM computing surveys (CSUR)*, vol. 31, no. 3, pp. 264–323, 1999.

[43] A. Ahmad and L. Dey, "A k-mean clustering algorithm for mixed numeric and categorical data," *Data & Knowledge Engineering*, vol. 63, no. 2, pp. 503–527, 2007.

[44] K. S. Al-Sultana and M. M. Khan, "Computational experience on four algorithms for the hard clustering problem," *Pattern recognition letters*, vol. 17, no. 3, pp. 295–308, 1996.

[45] K. Sanse and M. Sharma, "Clustering methods for big data analysis," *International Journal of Advanced Research in Computer Engineering & Technology*, vol. 4, no. 3, pp. 642–648, 2015.

[46] H. S. Deshmukh and P. Ramteke, "Comparing the techniques of cluster analysis for big data," *Int. J. Adv. Res. Comput. Eng. Technol*, vol. 4, no. 12, 2015.

[47] R. Suganya, M. Pavithra, and P. Nandhini, "Algorithms and challenges in big data clustering," *International Journal of Engineering and Techniques*, vol. 4, no. 4, pp. 40–47, 2018.

[48] J. A. Hartigan and M. A. Wong, "Algorithm as 136: A k-means clustering algorithm," *Journal of the royal statistical society. series c (applied statistics)*, vol. 28, no. 1, pp. 100–108, 1979.

[49] J. MacQueen *et al.*, "Some methods for classification and analysis of multivariate observations," in *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, Oakland, CA, USA, vol. 1, 1967, pp. 281–297.

[50] P. Rathore, "Big data cluster analysis and its applications.," Ph.D. dissertation, University of Melbourne, Parkville, Victoria, Australia, 2018.

[51] G. Hamerly and C. Elkan, "Learning the k in k-means," *Advances in neural information processing systems*, vol. 16, 2003.

[52] D. Pelleg, "Extending k-means with efficient estimation of the number of clusters in icml," in *Proceedings of the 17th international conference on machine learning*, 2000, pp. 277–281.

[53] C. Fraley and A. E. Raftery, "How many clusters? which clustering method? answers via model-based cluster analysis," *The computer journal*, vol. 41, no. 8, pp. 578–588, 1998.

[54] A. Dasgupta and A. E. Raftery, "Detecting features in spatial point processes with clutter via model-based clustering," *Journal of the American statistical Association*, vol. 93, no. 441, pp. 294–302, 1998.

[55] S. Mukherjee, E. D. Feigelson, G. J. Babu, F. Murtagh, C. Fraley, and A. Raftery, "Three types of gamma-ray bursts," *The Astrophysical Journal*, vol. 508, no. 1, p. 314, 1998.

[56] J. G. Campbell, C. Fraley, F. Murtagh, and A. E. Raftery, "Linear flaw detection in woven textiles using model-based clustering," *Pattern Recognition Letters*, vol. 18, no. 14, pp. 1539–1548, 1997.

[57] A. P. Dempster, N. M. Laird, and D. B. Rubin, "Maximum likelihood from incomplete data via the em algorithm," *Journal of the royal statistical society: series B (methodological)*, vol. 39, no. 1, pp. 1–22, 1977.

[58] G. J. McLachlan and T. Krishnan, *Wiley series in probability and statistics. the em algorithm and extensions*, 1997.

[59] B. Shekar, M. N. Murty, and G. Krishna, "A knowledge-based clustering scheme," *Pattern recognition letters*, vol. 5, no. 4, pp. 253–259, 1987.

[60] R. J. Trudeau, *Introduction to graph theory*. Courier Corporation, 2013.

[61] P. Berkhin, J. D. Beche, and D. J. Randall, "Interactive path analysis of web site traffic," in *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, 2001, pp. 414–419.

[62] J.-S. Cherng and M.-J. Lo, "A hypergraph based clustering algorithm for spatial data sets," in *Proceedings 2001 IEEE International Conference on Data Mining*, IEEE, 2001, pp. 83–90.

[63] C. T. Zahn, "Graph-theoretical methods for detecting and describing gestalt clusters," *IEEE Transactions on computers*, vol. 100, no. 1, pp. 68–86, 1971.

[64] R. Sharan and R. Shamir, "Click: A clustering algorithm with applications to gene expression analysis," in *Proc Int Conf Intell Syst Mol Biol*, Maryland, MD, vol. 8, 2000, p. 16.

[65] S. W. Wharton, "A generalized histogram clustering scheme for multidimensional image data," *Pattern Recognition*, vol. 16, no. 2, pp. 193–199, 1983.

[66] L. Parsons, E. Haque, and H. Liu, "Subspace clustering for high dimensional data: A review," *Acm sigkdd explorations newsletter*, vol. 6, no. 1, pp. 90–105, 2004.

[67] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan, "Automatic subspace clustering of high dimensional data for data mining applications," in *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, 1998, pp. 94–105.

[68] C.-H. Cheng, A. W. Fu, and Y. Zhang, "Entropy-based subspace clustering for mining numerical data," in *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*, 1999, pp. 84–93.

[69] H. Nagesh, S. Goil, and A. Choudhary, "Mafia: Efficient and scalable subspace clustering for very large data sets," *Technical Report 9906–010*, 1999.

[70] J.-W. Chang and D.-S. Jin, "A new cell-based clustering method for large, high-dimensional data in data mining applications," in *Proceedings of the 2002 ACM symposium on Applied computing*, 2002, pp. 503–507.

[71] C. M. Procopiuc, M. Jones, P. K. Agarwal, and T. Murali, "A monte carlo algorithm for fast projective clustering," in *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, 2002, pp. 418–427.

[72] C. C. Aggarwal, J. L. Wolf, P. S. Yu, C. Procopiuc, and J. S. Park, "Fast algorithms for projected clustering," *ACM SIGMoD record*, vol. 28, no. 2, pp. 61–72, 1999.

[73] C. C. Aggarwal, A. Hinneburg, and D. A. Keim, "On the surprising behavior of distance metrics in high dimensional space," in *Database Theory—ICDT 2001: 8th International Conference London, UK, January 4–6, 2001 Proceedings 8*, Springer, 2001, pp. 420–434.

[74] K.-G. Woo, J.-H. Lee, M.-H. Kim, and Y.-J. Lee, "Findit: A fast and intelligent subspace clustering algorithm using dimension voting," *Information and Software Technology*, vol. 46, no. 4, pp. 255–271, 2004.

[75] J. H. Friedman and J. J. Meulman, "Clustering objects on subsets of attributes (with discussion)," *Journal of the Royal Statistical Society Series B: Statistical Methodology*, vol. 66, no. 4, pp. 815–849, 2004.

[76] J. Yang, W. Wang, H. Wang, and P. Yu, "/spl delta/-clusters: Capturing subspace correlation in a large data set," in *Proceedings 18th international conference on data engineering*, IEEE, 2002, pp. 517–528.

[77] A. José-Garcıa and W. Gómez-Flores, "Automatic clustering using nature-inspired metaheuristics: A survey," *Applied Soft Computing*, vol. 41, pp. 192–213, 2016.

[78] Z. Aliniya and S. A. Mirroshandel, "A novel combinatorial merge-split approach for automatic clustering using imperialist competitive algorithm," *Expert Systems with Applications*, vol. 117, pp. 243–266, 2019.

[79] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan, "Automatic subspace clustering of high dimensional data," *Data Mining and Knowledge Discovery*, vol. 11, pp. 5–33, 2005.

[80] P. Agarwal, M. A. Alam, and R. Biswas, "Issues, challenges and tools of clustering algorithms," *arXiv preprint arXiv:1110.2610*, 2011.

[81] R.-J. Kuo, Y. Huang, C.-C. Lin, Y.-H. Wu, and F. E. Zulvia, "Automatic kernel clustering with bee colony optimization algorithm," *Information Sciences*, vol. 283, pp. 107–122, 2014.

[82] E. Falkenauer, *Genetic algorithms and grouping problems*. John Wiley & Sons, Inc., 1998.

[83] A. E. Ezugwu, A. K. Shukla, M. B. Agbaje, O. N. Oyelade, A. José-Garcıa, and J. O. Agushaka, "Automatic clustering algorithms: A systematic review and bibliometric analysis of relevant literature," *Neural Computing and Applications*, vol. 33, pp. 6247–6306, 2021.

[84] H. He and Y. Tan, "A two-stage genetic algorithm for automatic clustering," *Neurocomputing*, vol. 81, pp. 49–59, 2012.

[85] D. Doval, S. Mancoridis, and B. S. Mitchell, "Automatic clustering of software systems using a genetic algorithm," in *STEP'99. Proceedings Ninth International Workshop Software Technology and Engineering Practice*, IEEE, 1999, pp. 73–81.

[86] S. Paterlini and T. Krink, "Differential evolution and particle swarm optimisation in partitional clustering," *Computational statistics & data analysis*, vol. 50, no. 5, pp. 1220–1247, 2006.

[87] K. Suresh, D. Kundu, S. Ghosh, S. Das, and A. Abraham, "Data clustering using multi-objective differential evolution algorithms," *Fundamenta Informaticae*, vol. 97, no. 4, pp. 381–403, 2009.

[88] Z.-g. Su, P.-h. Wang, J. Shen, Y.-g. Li, Y.-f. Zhang, and E.-j. Hu, "Automatic fuzzy partitioning approach using variable string length artificial bee colony (vabc) algorithm," *Applied soft computing*, vol. 12, no. 11, pp. 3421–3441, 2012.

[89] Z. Izakian, M. S. Mesgari, and A. Abraham, "Automated clustering of trajectory data using a particle swarm optimization," *Computers, Environment and Urban Systems*, vol. 55, pp. 55–65, 2016.

[90] A. Abraham, S. Das, and S. Roy, "Swarm intelligence algorithms for data clustering," in *Soft computing for knowledge discovery and data mining*, Springer, 2008, pp. 279–313.

[91] A. Chowdhury, S. Bose, and S. Das, "Automatic clustering based on invasive weed optimization algorithm," in *Swarm, Evolutionary, and Memetic Computing: Second International Conference, SEMCCO 2011, Visakhapatnam, Andhra Pradesh, India, December 19-21, 2011, Proceedings, Part II 2*, Springer, 2011, pp. 105–112.

[92] S. Das, A. Chowdhury, and A. Abraham, "A bacterial evolutionary algorithm for automatic data clustering," in *2009 IEEE congress on evolutionary computation*, IEEE, 2009, pp. 2403–2410.

[93] J. Senthilnath, S. Omkar, and V. Mani, "Clustering using firefly algorithm: Performance study," *Swarm and Evolutionary Computation*, vol. 1, no. 3, pp. 164–171, 2011.

[94] J. H. Holland, *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.

[95] R. H. Sheikh, M. M. Raghuwanshi, and A. N. Jaiswal, "Genetic algorithm based clustering: A survey," in *2008 first international conference on emerging trends in engineering and technology*, IEEE, 2008, pp. 314–319.

[96] D. E. Goldberg, "Cenetic algorithms in search," *Optimization, Machine Learning*, 1989.

[97] R. Krovi, "Genetic algorithms for clustering: A preliminary investigation," in *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences*, IEEE, vol. 4, 1992, pp. 540–544.

[98] K. Krishna and M. N. Murty, "Genetic k-means algorithm," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 29, no. 3, pp. 433–439, 1999.

[99] Y. Lu, S. Lu, F. Fotouhi, Y. Deng, and S. J. Brown, "Fgka: A fast genetic k-means clustering algorithm," in *Proceedings of the 2004 ACM symposium on Applied computing*, 2004, pp. 622–623.

[100] Y. Lu, S. Lu, F. Fotouhi, Y. Deng, and S. J. Brown, "Incremental genetic k-means algorithm and its application in gene expression data analysis," *BMC bioinformatics*, vol. 5, no. 1, pp. 1–10, 2004.

[101] U. Maulik and S. Bandyopadhyay, "Genetic algorithm-based clustering technique," *Pattern recognition*, vol. 33, no. 9, pp. 1455–1465, 2000.

[102] L. O. Hall, I. B. Ozyurt, and J. C. Bezdek, "Clustering with a genetically optimized approach," *IEEE Transactions on evolutionary Computation*, vol. 3, no. 2, pp. 103–112, 1999.

[103] M. Dorigo, M. Birattari, and T. Stutzle, "Ant colony optimization," *IEEE computational intelligence magazine*, vol. 1, no. 4, pp. 28–39, 2006.

[104] J.-L. Deneubourg, S. Goss, N. Franks, A. Sendova-Franks, C. Detrain, and L. Chrétien, "The dynamics of collective sorting robot-like ants and ant-like robots," in *From animals to animats: proceedings of the first international conference on simulation of adaptive behavior*, 1991, pp. 356–365.

[105] M. Dorigo, V. Maniezzo, and A. Colorni, "Ant system: Optimization by a colony of cooperating agents," *IEEE transactions on systems, man, and cybernetics, part b (cybernetics)*, vol. 26, no. 1, pp. 29–41, 1996.

[106] T. A. Runkler, "Ant colony optimization of clustering models," *International Journal of Intelligent Systems*, vol. 20, no. 12, pp. 1233–1251, 2005.

[107] S. Saatchi and C. C. Hung, "Hybridization of the ant colony optimization with the k-means algorithm for clustering," in *Image Analysis: 14th Scandinavian Conference, SCIA 2005, Joensuu, Finland, June 19-22, 2005. Proceedings 14*, Springer, 2005, pp. 511–520.

[108] P. M. Kanade and L. O. Hall, "Fuzzy ant clustering by centroid positioning," in *2004 IEEE International Conference on Fuzzy Systems (IEEE Cat. No. 04CH37542)*, IEEE, vol. 1, 2004, pp. 371–376.

[109] J. Handl, J. Knowles, and M. Dorigo, "Ant-based clustering and topographic mapping," *Artificial life*, vol. 12, no. 1, pp. 35–62, 2006.

[110] A. P. Engelbrecht, *Fundamentals of computational swarm intelligence*. John Wiley & Sons, Inc., 2006.

[111] J. Handl and J. Knowles, "An evolutionary approach to multiobjective clustering," *IEEE transactions on Evolutionary Computation*, vol. 11, no. 1, pp. 56–76, 2007.

[112] M. G. Omran, A. P. Engelbrecht, and A. Salman, "Image classification using particle swarm optimization," in *Recent advances in simulated evolution and learning*, World Scientific, 2004, pp. 347–365.

[113] S. Das, A. Abraham, and S. K. Sarkar, "A hybrid rough set–particle swarm algorithm for image pixel classification," in *2006 Sixth International Conference on Hybrid Intelligent Systems (HIS'06)*, IEEE, 2006, pp. 26–26.

[114] J. Qu, Z. Shao, X. Liu, *et al.*, "Mixed pso clustering algorithm using point symmetry distance," 2010.

[115] S. Ouadfel, M. Batouche, and A. Taleb-Ahmed, "A modified particle swarm optimization algorithm for automatic image clustering," in *International Symposium on Modelling and Implementation of Complex Systems, MISC*, vol. 2010, 2010.

[116] T. Cura, "A particle swarm optimization approach to clustering," *Expert Systems with Applications*, vol. 39, no. 1, pp. 1582–1588, 2012.

[117] R. Kuo, Y. Syu, Z.-Y. Chen, and F.-C. Tien, "Integration of particle swarm optimization and genetic algorithm for dynamic clustering," *Information Sciences*, vol. 195, pp. 124–140, 2012.

[118] D. Van der Merwe and A. P. Engelbrecht, "Data clustering using particle swarm optimization," in *The 2003 Congress on Evolutionary Computation, 2003. CEC'03.*, IEEE, vol. 1, 2003, pp. 215–220.

[119] X. Cui, T. E. Potok, and P. Palathingal, "Document clustering using particle swarm optimization," in *Proceedings 2005 IEEE Swarm Intelligence Symposium, 2005. SIS 2005.*, IEEE, 2005, pp. 185–191.

[120] X. Cui and T. E. Potok, "Document clustering analysis based on hybrid pso+ k-means algorithm," *Journal of Computer Sciences (special issue)*, vol. 27, p. 33, 2005.

[121] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. Piatko, R. Silverman, and A. Y. Wu, "The analysis of a simple k-means clustering algorithm," in *Proceedings of the sixteenth annual symposium on Computational geometry*, 2000, pp. 100–109.

[122] M. Aitkin and D. B. Rubin, "Estimation and hypothesis testing in finite mixture models," *Journal of the Royal Statistical Society Series B: Statistical Methodology*, vol. 47, no. 1, pp. 67–75, 1985.

[123] F. H. C. Marriott, "The interpretation of multiple observations," *(No Title)*, 1974.

[124] G. J. McLachlan and K. E. Basford, *Mixture models: Inference and applications to clustering*. M. Dekker New York, 1988, vol. 38.

[125] N. Grira, M. Crucianu, and N. Boujemaa, "Unsupervised and semi-supervised clustering: A brief survey," *A review of machine learning techniques for processing multimedia content*, vol. 1, no. 2004, pp. 9–16, 2004.

[126] J. Verbeek, "Mixture models for clustering and dimension reduction," Ph.D. dissertation, Universiteit van Amsterdam, 2004.

[127] C. F. Olson, "Parallel algorithms for hierarchical clustering," *Parallel computing*, vol. 21, no. 8, pp. 1313–1325, 1995.

[128] F. Murtagh, "A survey of algorithms for contiguity-constrained clustering and related problems," *The computer journal*, vol. 28, no. 1, pp. 82–88, 1985.

[129] R. Sibson, "Slink: An optimally efficient algorithm for the single-link cluster method," *The computer journal*, vol. 16, no. 1, pp. 30–34, 1973.

[130] D. Defays, "An efficient algorithm for a complete link method," *The Computer Journal*, vol. 20, no. 4, pp. 364–366, 1977.

[131] E. M. Voorhees, "Implementing agglomerative hierarchic clustering algorithms for use in document retrieval," *Information Processing & Management*, vol. 22, no. 6, pp. 465–476, 1986.

[132] J. H. Ward Jr, "Hierarchical grouping to optimize an objective function," *Journal of the American statistical association*, vol. 58, no. 301, pp. 236–244, 1963.

[133] K. C. Gowda and G. Krishna, "Agglomerative clustering using the concept of mutual nearest neighbourhood," *Pattern recognition*, vol. 10, no. 2, pp. 105–112, 1978.

[134] D. Boley, "Principal direction divisive partitioning," *Data mining and knowledge discovery*, vol. 2, pp. 325–344, 1998.

[135] S. M. Savaresi, D. L. Boley, S. Bittanti, and G. Gazzaniga, "Cluster selection in divisive clustering algorithms," in *Proceedings of the 2002 SIAM International Conference on Data Mining*, SIAM, 2002, pp. 299–314.

[136] M. Chavent, Y. Lechevallier, and O. Briant, "Divclus-t: A monothetic divisive hierarchical clustering method," *Computational Statistics & Data Analysis*, vol. 52, no. 2, pp. 687–701, 2007.

[137] G. Karypis and V. Kumar, "Multilevel k-way hypergraph partitioning," in *Proceedings of the 36th annual ACM/IEEE design automation conference*, 1999, pp. 343–348.

[138] L. Feng, M.-H. Qiu, Y.-X. Wang, Q.-L. Xiang, Y.-F. Yang, and K. Liu, "A fast divisive clustering algorithm using an improved discrete particle swarm optimizer," *Pattern Recognition Letters*, vol. 31, no. 11, pp. 1216–1225, 2010.

[139] C. Zhong, D. Miao, R. Wang, and X. Zhou, "Divfrp: An automatic divisive hierarchical clustering method based on the furthest reference points," *Pattern Recognition Letters*, vol. 29, no. 16, pp. 2067–2077, 2008.

[140] P. Macnaughton-Smith, W. Williams, M. Dale, and L. Mockett, "Dissimilarity analysis: A new technique of hierarchical sub-division," *Nature*, vol. 202, no. 4936, pp. 1034–1035, 1964.

[141] L. Kaufman and P. J. Rousseeuw, *Finding groups in data: an introduction to cluster analysis*. John Wiley & Sons, 2009.

[142] P. H. Sneath, R. R. Sokal, *et al.*, *Numerical taxonomy. The principles and practice of numerical classification.* 1973.

[143] W. T. Williams and J. M. Lambert, "Multivariate methods in plant ecology: I. association-analysis in plant communities," *The Journal of Ecology*, pp. 83–101, 1959.

[144] J. Kim, "Dissimilarity measures for histogram-valued data and divisive clustering of symbolic objects," Ph.D. dissertation, University of Georgia Athens, GA, USA, 2009.

[145] P. M. Brito and M. Chavent, "Divisive monothetic clustering for interval and histogram-valued data," in *ICPRAM 2012-1st International Conference on Pattern Recognition Applications and Methods*, 2012, pp. 229–234.

[146] J. Kim and L. Billard, "Dissimilarity measures and divisive clustering for symbolic multimodal-valued data," *Computational Statistics & Data Analysis*, vol. 56, no. 9, pp. 2795–2808, 2012.

[147] J. Kim and L. Billard, "A polythetic clustering process and cluster validity indexes for histogram-valued objects," *Computational Statistics & Data Analysis*, vol. 55, no. 7, pp. 2250–2262, 2011.

[148] S. Wang, J. Fan, M. Fang, and H. Yuan, "Hgcudf: Hierarchical grid clustering using data field," *Chinese Journal of Electronics*, vol. 23, no. 1, pp. 37–42, 2014.

[149] I. Naim, S. Datta, J. Rebhahn, J. S. Cavenaugh, T. R. Mosmann, and G. Sharma, "Swift—scalable clustering for automated identification of rare cell populations in large, high-dimensional flow cytometry datasets, part 1: Algorithm design," *Cytometry Part A*, vol. 85, no. 5, pp. 408–421, 2014.

[150] T. Herawan, M. M. Deris, and J. H. Abawajy, "A rough set approach for selecting clustering attribute," *Knowledge-Based Systems*, vol. 23, no. 3, pp. 220–231, 2010.

[151] L. Mazlack, A. He, Y. Zhu, and S. Coppock, "A rough set approach in choosing partitioning attributes," in *Proceedings of the ISCA 13th International Conference (CAINE-2000)*, Citeseer, 2000, pp. 1–6.

[152] D. Parmar, T. Wu, and J. Blackhurst, "Mmr: An algorithm for clustering categorical data using rough set theory," *Data & Knowledge Engineering*, vol. 63, no. 3, pp. 879–893, 2007.

[153] T. Herawan and M. M. Deris, "A framework on rough set-based partitioning attribute selection," in *Emerging Intelligent Computing Technology and Applications. With Aspects of Artificial Intelligence: 5th International Conference on Intelligent Computing, ICIC 2009 Ulsan, South Korea, September 16-19, 2009 Proceedings 5*, Springer, 2009, pp. 91–100.

[154] T. Xiong, S. Wang, A. Mayers, and E. Monga, "A new mca-based divisive hierarchical algorithm for clustering categorical data," in *2009 Ninth IEEE International Conference on Data Mining*, IEEE, 2009, pp. 1058–1063.

[155] H. Qin, X. Ma, T. Herawan, and J. M. Zain, "Mgr: An information theory based hierarchical divisive clustering algorithm for categorical data," *Knowledge-Based Systems*, vol. 67, pp. 401–411, 2014.

[156] K. Lo, R. R. Brinkman, and R. Gottardo, "Automated gating of flow cytometry data via robust model-based clustering," *Cytometry Part A: the journal of the International Society for Analytical Cytology*, vol. 73, no. 4, pp. 321–332, 2008.

[157] Y. Ge and S. C. Sealfon, "Flowpeaks: A fast unsupervised clustering for flow cytometry data via k-means and density peak finding," *Bioinformatics*, vol. 28, no. 15, pp. 2052–2058, 2012.

[158] G. Finak, A. Bashashati, R. Brinkman, and R. Gottardo, "Merging mixture components for cell population identification in flow cytometry," *Advances in bioinformatics*, vol. 2009, 2009.

[159] R. F. Murphy, "Automated identification of subpopulations in flow cytometric list mode data using cluster analysis," *Cytometry: The Journal of the International Society for Analytical Cytology*, vol. 6, no. 4, pp. 302–309, 1985.

[160] H. Zare, P. Shooshtari, A. Gupta, and R. R. Brinkman, "Data reduction for spectral clustering to analyze high throughput flow cytometry data," *BMC bioinformatics*, vol. 11, pp. 1–16, 2010.

[161] T. Zhang, R. Ramakrishnan, and M. Livny, "Birch: An efficient data clustering method for very large databases," *ACM sigmod record*, vol. 25, no. 2, pp. 103–114, 1996.

[162] D. Arthur and S. Vassilvitskii, "K-means++ the advantages of careful seeding," in *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, 2007, pp. 1027–1035.

[163] R. A. Fisher, *Iris*, UCI Machine Learning Repository, DOI: https://doi.org/10.24432/C56C76, 1988.

[164] F. Pedregosa, G. Varoquaux, A. Gramfort, *et al.*, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

Appendix A

Source Code Listings

A.1 K-Means

```
kmeans <- function(
  data,
  centers,
  max_iterations = 10,
  initialization = "kmeans++",
  details = FALSE,
  waiting = TRUE,
  ...
) {
  # Make sure max_iterations is a positive integer
  if (!is.numeric(max_iterations) || max_iterations < 1)
    stop("max_iterations must be an integer greater than 0")

  # Get centers
  if (missing(centers))
    stop("centers must be a matrix or a number")

  init <- 3
  if (length(centers) == 1) {
    if (centers < 1)
      stop("centers must be a positive integer")
    if (centers > nrow(data))
      stop("centers must be less than or equal to the number of observations")

    # Figure out the initialization method
    init <- grep(
      tolower(initialization),
      c("random", "kmeans++"),
      fixed = TRUE
    )

    if (length(init) != 1)
      stop("initialization must be one of 'random' or 'kmeans++'")
  }

  if (details) {
    hline()
    console.log("EXPLANATION:")
    console.log("")
    console.log("The K-Means algorithm aims to partition a dataset into k groups such
```

```
41          that the within-cluster sum-of-squares is minimized. At the minimum, all cluster
42          centers are at the mean of their Voronoi sets (the set of data points which are
43          nearest to the cluster center).")
44        console.log("")
45        console.log("The K-Means method follows a 2 to n step process:")
46        console.log("")
47        console.log("    1. The first step can be subdivided into 3 steps:")
48        console.log("        1. Selection of the number k of clusters, into which the data
49            is going to be grouped and of which the centroids will be the representatives.")
50        console.log("        2. Computation of the distance from each observation to each
51            centroid.")
52        console.log("        3. Assignment of each observation to a cluster. Observations
53            are assigned to the cluster represented by the nearest centroid.")
54        console.log("    2. The next steps are just like the first but for the first sub-step
55            we do the following:")
56        console.log("        1. Computation of the new centroids The centroid of each cluster
57            is computed as the mean of the observations assigned to said cluster.")
58        console.log("")
59        console.log("The algorithm stops once the centers in step n+1 are the same as the ones
60            in step n. However, this convergence does not always take place. For this reason,
61            the algorithm also stops once a maximum number of iterations is reached.")
62        console.log("")
63
64        if (waiting) {
65          invisible(readline(prompt = "Press [enter] to continue"))
66          console.log("")
67        }
68
69        hline()
70        console.log("STEP: 1")
71        console.log("")
72        console.log("If they are not, k centroids have to be initialized...")
73        console.log("")
74      }
75
76      # Initialize centers ...
77      centers <- switch(
78        init,
79        # ... randomly
80        random_init(as.matrix(data), centers, details, waiting, ...),
81        # ... using the kmeans++ algorithm
82        kmeanspp_init(as.matrix(data), centers, details, waiting, ...),
83        # ... they are already initialized
84        centers
85      )
86
87      if (details) {
88        console.log("With this, the k initial centroids are the following:")
89        cat("Centroids:\n")
90        print(centers)
91        console.log("")
92
93        if (waiting) {
94          invisible(readline(prompt = "Press [enter] to continue"))
95          console.log("")
96        }
97      }
98
99      # Update centers while they don't change
100     iter <- 0
101     for (i in seq_len(max_iterations)) {
```

```
102       iter <- i
103       old_centers <- as.matrix(centers)
104
105       # Compute distances between points and centers
106       distances <- proxy::dist(old_centers, data, ...)
107
108       if (details) {
109         console.log("With these centroids, the matrix of pairwise distances between the
110           observations and the centroids is computed:")
111         cat("Distances:\n")
112         print(round(distances, 3))
113         console.log("")
114
115         if (waiting) {
116           invisible(readline(prompt = "Press [enter] to continue"))
117           console.log("")
118         }
119       }
120
121       # Find which center is closest to each point
122       nearest_centers <- apply(distances, 2, which.min)
123
124       if (details) {
125         console.log("From these distances, the centroid closest to each observation is
126           computed. In this way, we make the following observation-cluster assignments:")
127         cat("Cluster assignments:\n")
128         print(nearest_centers)
129         console.log("")
130
131         if (waiting) {
132           invisible(readline(prompt = "Press [enter] to continue"))
133           console.log("")
134         }
135       }
136
137       # Compute the new centers as the average of it's closest points
138       new_centers <- matrix(
139         sapply(
140           seq_len(nrow(old_centers)),
141           function(n) {
142             temp <- data[nearest_centers == n, , drop = FALSE]
143             if (nrow(temp) > 0)
144               apply(temp, 2, mean)
145             else
146               old_centers[n, ]
147           }
148         ),
149         nrow = ncol(old_centers)
150       )
151       centers <- t(new_centers)
152
153       if (details) {
154         hline()
155         console.log(paste("STEP:", iter + 1))
156         console.log("")
157         console.log("With the previous cluster assignments, we compute the new centroids
158           as the mean of the observations assigned to the corresponding cluster:")
159         cat("Centroids:\n")
160         print(centers)
161         console.log("")
162
```

```
163       if (waiting) {
164         invisible(readline(prompt = "Press [enter] to continue"))
165         console.log("")
166       }
167     }
168
169     # If centers aren't updated
170     if (all(centers == old_centers))
171       break
172   }
173
174   # Compute distances between points and centers
175   distances <- proxy::dist(centers, data, ...)
176
177   if (details) {
178     if (iter == max_iterations)
179       console.log("Since we have reached the maximum amount of iterations, these are
180         the last centroids we are going to compute.")
181     else
182       console.log("Since the centroids have not changed with regards to last step,
183         these are the last centroids we are going to compute.")
184     console.log("With these centroids, the matrix of pairwise distances between the
185         observations and the centroids is computed one last time:")
186     cat("Distances:\n")
187     print(round(distances, 3))
188     console.log("")
189
190     if (waiting) {
191       invisible(readline(prompt = "Press [enter] to continue"))
192       console.log("")
193     }
194   }
195
196   # Find which center is closest to each point
197   nearest_centers <- apply(distances, 2, which.min)
198   row.names(centers) <- seq_len(nrow(centers))
199
200   if (details) {
201     console.log("From these distances, observations are assigned the cluster of whichever
202         centroid they are closest to:")
203     cat("Cluster assignments:\n")
204     print(nearest_centers)
205     console.log("")
206
207     if (waiting) {
208       invisible(readline(prompt = "Press [enter] to continue"))
209       console.log("")
210     }
211
212     hline()
213     console.log("")
214   }
215
216   # Total sum of squares
217   center <- apply(data, 2, mean)
218   totss <- sum(apply(data, 1, function(x) x - center)^2)
219
220   # Total within-cluster sum of squares
221   withinss <- sapply(
222     seq_len(nrow(centers)),
223     function(cluster) {
```

```
224         ccenter <- centers[cluster, ]
225         cdata <- data[nearest_centers == cluster, , drop = FALSE]
226         sum(apply(cdata, 1, function(x) x - ccenter)^2)
227       }
228     )
229
230     # Total within-cluster sum of squares
231     tot.withinss <- sum(withinss)
232
233     # The between-cluster sum of squares
234     betweenss <- totss - tot.withinss
235
236     # Find the size of each cluster
237     tmp <- factor(nearest_centers, levels = seq_len(nrow(centers)))
238     size <- as.integer(table(tmp))
239
240     structure(
241       list(
242         cluster = nearest_centers,
243         centers = centers,
244         totss = totss,
245         withinss = withinss,
246         tot.withinss = tot.withinss,
247         betweenss = betweenss,
248         size = size,
249         iter = iter,
250         ifault = 0
251       ),
252       class = "kmeans"
253     )
254   }
255
256   random_init <- function(data, k, details, waiting, ...) {
257     if (details) {
258       console.log("In this case, the k centroids are chosen randomly from the observations...")
259       console.log("")
260     }
261
262     smp <- sample(nrow(data), size = k, replace = FALSE)
263     data[smp, , drop = FALSE]
264   }
265
266   kmeanspp_init <- function(data, k, details, waiting, ...) {
267     if (details) {
268       console.log("In this case, the k centroids are chosen using the kmeans++ algorithm...")
269       console.log("")
270     }
271
272     centers <- matrix(0, nrow = k, ncol = ncol(data))
273     probs <- rep(1 / nrow(data), nrow(data))
274     for (i in seq_len(k)) {
275       # Choose a center with probability proportional to its square distance to
276       # the closest center
277       smp <- sample(nrow(data), size = 1, replace = FALSE, prob = probs)
278       centers[i, ] <- data[smp, ]
279
280       if (details) {
281         console.log(paste0("--- kmeans++ step #", i, " ---\n"))
282         console.log("A centroid is chosen according to the following probabilities:")
283         cat("Probs:\n")
284         print(round(probs, 3))
```

```
285         console.log("The chosen centroid is:")
286         cat(paste0("Observation #", smp, ":\n"))
287         print(centers[i, ])
288         cat("With probability:\n")
289         print(probs[[smp]])
290         if (i < k)
291           console.log("With this new centroid the probabilities are updated...")
292         else
293           console.log("With this new centroid we already have k centroids...")
294         console.log("")
295 
296         if (waiting) {
297           invisible(readline(prompt = "Press [enter] to continue"))
298           console.log("")
299         }
300       }
301 
302       # Update the probabilities
303       distances <- proxy::dist(centers[seq_len(i), , drop = FALSE], data, ...) ^ 2
304       probs <- apply(distances, 2, min)
305       probs <- probs / sum(probs)
306 
307       # Replace NAs and NaNs with the remaining probability
308       tmp <- sum(probs[is.finite(probs)])
309       probs[!is.finite(probs)] <- (1 - tmp) / sum(!is.finite(probs))
310     }
311     centers
312   }
```

Code A.1: R implementation of the K-Means algorithm

A.2 DBSCAN

```
1   dbscan <- function(
2     data,
3     eps,
4     min_pts = 4,
5     details = FALSE,
6     waiting = TRUE,
7     ...
8   ) {
9     if (details) {
10      hline()
11      console.log("EXPLANATION:")
12      console.log("")
13      console.log("The data given by data is clustered by the DBSCAN method, which aims
14          to partition the points into clusters such that the points in a cluster are close
15          to each other and the points in different clusters are far away from each other.
16          The clusters are defined as dense regions of points separated by regions of low
17          density.")
18      console.log("")
19      console.log("The DBSCAN method follows a 2 step process:")
20      console.log("")
21      console.log("    1. For each point, the neighborhood of radius eps is computed. If
22          the neighborhood contains at least min_pts points, then the point is considered a
23          core point. Otherwise, the point is considered an outlier.")
24      console.log("    2. For each core point, if the core point is not already assigned
25          to a cluster, a new cluster is created and the core point is assigned to it. Then,
```

```
26          the neighborhood of the core point is explored. If a point in the neighborhood is
27          a core point, then the neighborhood of that point is also explored. This process
28          is repeated until all points in the neighborhood have been explored. If a point
29          in the neighborhood is not already assigned to a cluster, then it is assigned to
30          the cluster of the core point.")
31        console.log("")
32        console.log("Whatever points are not assigned to a cluster are considered outliers.")
33        console.log("")
34
35        if (waiting) {
36          invisible(readline(prompt = "Press [enter] to continue"))
37          console.log("")
38        }
39      }
40
41      # Precompute neighbors
42      distances <- as.matrix(proxy::dist(data, ...))
43      neighbors <- distances <= eps
44
45      if (details) {
46        hline()
47        console.log("STEP 1:")
48        console.log("")
49        console.log("The pairwise distances between observations are precomputed in order to
50            later determine which of them are core observations. The distance matrix is:")
51        cat("Distances:\n")
52        print(round(distances, 3))
53        console.log("")
54
55        if (waiting) {
56          invisible(readline(prompt = "Press [enter] to continue"))
57          console.log("")
58        }
59      }
60
61      # Initialize clusters
62      cluster_id <- new.env()
63      cluster_id$data <- data
64      cluster_id$current <- 1
65      cluster_id$of <- rep(-1, nrow(data))
66
67      if (details) {
68        hline()
69        console.log("STEP 2:")
70        console.log("")
71        console.log("Every observation is labeled as UNVISITED. We are now going to loop over
72            every observation and, if it is not already assigned to a cluster, we will try to
73            expand a new cluster around it...")
74        console.log("")
75
76        if (waiting) {
77          invisible(readline(prompt = "Press [enter] to continue"))
78          console.log("")
79        }
80      }
81
82      # Each loop finds a new cluster around a core point
83      for (idx in seq_len(nrow(data))) {
84        if (cluster_id$of[idx] != -1)
85          next
86        if (expand_cluster(neighbors, cluster_id, idx, min_pts, details, waiting))
```

```
87          cluster_id$current <- cluster_id$current + 1
88        }
89
90        if (details) {
91          hline()
92          console.log("RESULTS:")
93          console.log("")
94          console.log("Having gone through every observation the following clusters have been found:")
95          cat("CLUSTER #0 (NOISE):\n")
96          print(cluster_id$data[cluster_id$of == 0, ])
97          for (i in seq_len(max(cluster_id$of))) {
98            console.log("")
99            cat(paste0("CLUSTER #", i, ":\n"))
100           print(cluster_id$data[cluster_id$of == i, ])
101         }
102         console.log("")
103
104         if (waiting) {
105           invisible(readline(prompt = "Press [enter] to continue"))
106           console.log("")
107         }
108         hline()
109       }
110
111       # Compute cluster sizes
112       size <- as.integer(table(cluster_id$of))
113
114       # Return a dbscan object
115       structure(
116         list(
117           cluster = cluster_id$of,
118           eps = eps,
119           min_pts = min_pts,
120           size = size
121         ),
122         class = "dbscan"
123       )
124     }
125
126     expand_cluster <- function(
127       neighbors,
128       cluster_id,
129       point,
130       min_pts,
131       details,
132       waiting
133     ) {
134       # Get the point's neighbors (including itself)
135       seeds <- region_query(neighbors, point)
136
137       if (length(seeds) < min_pts) {
138         if (details) {
139           hline()
140           console.log("NOISE:")
141           console.log("")
142           console.log("An UNVISITED observation is labeled as NOISE:")
143           cat(paste0("Observation #", point, " [UNVISITED -> NOISE]\n"))
144           # print(cluster_id$data[point, ])
145           console.log("")
146
147           # console.log("With neighborhood:")
```

```
148         # for (i in seeds) {
149         #   tmp <- if (cluster_id$of[i] == -1) {
150         #     "UNVISITED"
151         #   } else if (cluster_id$of[i] == 0) {
152         #     "NOISE"
153         #   } else {
154         #     paste0("CLUSTER #", cluster_id$of[i])
155         #   }
156         #   ifelse(cluster_id$of[i] == -1, "UNVISITED", "NOISE")
157         #   cat(paste0("Observation #", i, " [", tmp, "]\n"))
158         #   print(cluster_id$data[i, ])
159         # }
160         # console.log("")
161
162         if (waiting) {
163           invisible(readline(prompt = "Press [enter] to continue"))
164           console.log("")
165         }
166       }
167
168       # If it is not a core point, it is noise
169       cluster_id$of[point] <- 0
170       FALSE
171     } else {
172       if (details) {
173         hline()
174         console.log(paste0("CLUSTER #", cluster_id$current, ":"))
175         console.log("")
176         console.log("A new cluster is going to be expanded around an UNVISITED core observation:")
177         cat(paste0("Observation #", point, " [UNVISITED -> CLUSTER #", cluster_id$current, "]\n"))
178         # print(cluster_id$data[point, ])
179         console.log("")
180
181         # console.log("With neighborhood:")
182         # for (i in seeds) {
183         #   tmp <- if (cluster_id$of[i] == -1) {
184         #     "UNVISITED"
185         #   } else if (cluster_id$of[i] == 0) {
186         #     "NOISE"
187         #   } else {
188         #     paste0("CLUSTER #", cluster_id$of[i])
189         #   }
190         #   ifelse(cluster_id$of[i] == -1, "UNVISITED", "NOISE")
191         #   cat(paste0("Observation #", i, " [", tmp, "]\n"))
192         #   print(cluster_id$data[i, ])
193         # }
194         # console.log("")
195
196         if (waiting) {
197           invisible(readline(prompt = "Press [enter] to continue"))
198           console.log("")
199         }
200       }
201
202       # Otherwise, we can expand the cluster
203       frontier <- setdiff(seeds, point)
204
205       if (details) {
206         console.log("The cluster is also expanded around the neighbors of the core observation:")
207         for (i in frontier) {
208           tmp1 <- ifelse(cluster_id$of[i] == -1, "UNVISITED", "NOISE")
```

```
209           tmp2 <- paste0("CLUSTER #", cluster_id$current)
210           cat(paste0("Observation #", i, " [", tmp1, " -> ", tmp2, "]\n"))
211           # print(cluster_id$data[i, ])
212         }
213         console.log("")
214
215         console.log("All of these observations are added to the cluster.")
216         console.log("")
217
218         if (waiting) {
219           invisible(readline(prompt = "Press [enter] to continue"))
220           console.log("")
221         }
222       }
223
224       cluster_id$of[seeds] <- cluster_id$current
225
226       # Loop until there are no more neighbors in the frontier
227       while (length(frontier) > 0) {
228         current_point <- frontier[1]
229
230         # Get current_point's neighbors
231         result <- region_query(neighbors, current_point)
232
233         # If current_point is a core point, expand the cluster
234         if (length(result) >= min_pts) {
235           # Add non visited neighbors to the frontier
236           not_visited <- cluster_id$of[result] == -1
237           frontier <- c(frontier, result[not_visited])
238
239           # Add not clustered neighbors to the cluster
240           noise <- cluster_id$of[result] == 0
241
242           if (details) {
243             console.log("***")
244             console.log("")
245             console.log("The following core observation is expanded:")
246             cat(paste0("Observation #", current_point, " [CLUSTER #", cluster_id$current, "]\n"))
247             # print(cluster_id$data[current_point, ])
248             console.log("")
249
250             console.log("It's neighborhood is:")
251             for (i in result) {
252               tmp <- if (cluster_id$of[i] == -1) {
253                 "UNVISITED"
254               } else if (cluster_id$of[i] == 0) {
255                 "NOISE"
256               } else {
257                 paste0("CLUSTER #", cluster_id$of[i])
258               }
259               ifelse(cluster_id$of[i] == -1, "UNVISITED", "NOISE")
260               cat(paste0("Observation #", i, " [", tmp, "]\n"))
261               # print(cluster_id$data[i, ])
262             }
263             console.log("")
264
265             if (length(result[not_visited | noise]) > 0) {
266               console.log("Upon doing it, the following observations are added to the cluster:")
267               for (i in result[not_visited | noise]) {
268                 tmp1 <- ifelse(cluster_id$of[i] == -1, "UNVISITED", "NOISE")
269                 tmp2 <- paste0("CLUSTER #", cluster_id$current)
```

```
270                 cat(paste0("Observation #", i, " [", tmp1, " -> ", tmp2, "]\n"))
271                 # print(cluster_id$data[i, ])
272               }
273             } else {
274               console.log("Upon doing it, no observations are added to the cluster...")
275             }
276             console.log("")
277
278             if (length(result[not_visited]) > 0) {
279               console.log("Additionally, these observations are also expanded:")
280               for (i in result[not_visited]) {
281                 cat(paste0("Observation #", i, " [CLUSTER #", cluster_id$current, "]\n"))
282                 # print(cluster_id$data[i, ])
283               }
284             } else {
285               console.log("Additionally, no other observations are expanded...")
286             }
287             console.log("")
288
289             if (waiting) {
290               invisible(readline(prompt = "Press [enter] to continue"))
291               console.log("")
292             }
293           }
294
295           cluster_id$of[result][not_visited | noise] <- cluster_id$current
296         }
297
298         # Remove current_point from the frontier
299         frontier <- frontier[-1]
300       }
301       TRUE
302     }
303   }
304
305   region_query <- function(
306     neighbors,
307     idx
308   ) {
309     # Return the indices of the neighbors of idx
310     which(neighbors[idx, ])
311   }
```

Code A.2: R implementation of the DBSCAN algorithm

A.3 Gaussian Mixture Model with Expectation Maximization

```
1   gaussian_mixture <- function(
2     data,
3     k,
4     max_iter = 10,
5     details = FALSE,
6     waiting = TRUE,
7     ...
8   ) {
9     data <- as.matrix(data)
10
11    if (details) {
```

```
12      hline()
13      console.log("EXPLANATION:")
14      console.log("")
15      console.log("The Gaussian Mixture Model with Expectation Maximization (GMM with EM)
16          algorithm aims to model the data as a Gaussian Mixture Model i.e. the weighted
17          sum of several Gaussian distributions, where each component i.e. each Gaussian
18          distribution, represents a cluster.")
19      console.log("")
20      console.log("The Gaussian distributions are parameterized by their mean vector (mu),
21          covariance matrix (sigma) and mixing proportion (lambda). Initially, the mean
22          vector is set to the cluster centers obtained by performing a k-means clustering
23          on the data, the covariance matrix is set to the covariance matrix of the data
24          points belonging to each cluster and the mixing proportion is set to the proportion
25          of data points belonging to each cluster. The algorithm then optimizes the GMM
26          using the EM algorithm.")
27      console.log("")
28      console.log("The EM algorithm is an iterative algorithm that alternates between two steps:")
29      console.log("")
30      console.log("    1. Expectation step. Compute how much is each observation expected
31          to belong to each component of the GMM.")
32      console.log("    2. Maximization step. Recompute the GMM according to the expectations
33          from the E-step in order to maximize them.")
34      console.log("")
35      console.log("The algorithm stops when the changes in the expectations are sufficiently
36          small or when a maximum number of iterations is reached.")
37      console.log("")
38
39      if (waiting) {
40        invisible(readline(prompt = "Press [enter] to continue"))
41        console.log("")
42      }
43    }
44
45    # Perform k-means to get initial values for mu, sigma and pi
46    members <- kmeans(data, k, ...)
47    mu <- members$centers
48    sigma <- array(0, dim = c(k, ncol(data), ncol(data)))
49    for (i in seq_len(k)) {
50      sigma[i, , ] <- as.matrix(cov(data[members$cluster == i, , drop = FALSE]))
51    }
52    lambda <- members$size / nrow(data)
53
54    if (details) {
55      hline()
56      console.log("INITIALIZATION:")
57      console.log("")
58      console.log("The GMM is initialized by calling kmeans. The initial components are:")
59      for (i in seq_len(k)) {
60        console.log(paste0("*** Component #", i, " ***\n"))
61        cat("mu:\n")
62        print(mu[i, ])
63        cat("sigma:\n")
64        print(as.matrix(sigma[i, , ]))
65        cat("lambda:\n")
66        print(lambda[i])
67        console.log("")
68      }
69      console.log("These initial components are then optimized using the EM algorithm.")
70      console.log("")
71
72      if (waiting) {
```

```
      invisible(readline(prompt = "Press [enter] to continue"))
      console.log("")
    }
  }

  # EM algorithm
  # Starting values of expected value of the log likelihood
  q <- c(
    sum_finite(
      sapply(
        seq_len(k),
        function(i) {
          log(lambda[i]) + log(dmnorm(data, mu[i, ], as.matrix(sigma[i, , ])))
        }
      )
    ),
    0
  )
  it <- 0
  if (details) {
    hline()
    console.log("EM ALGORITHM:")
    console.log("")
    console.log("To measure how much the expectations change at each step we will
        use the log likelihood. The log likelihood is the sum of the logarithm of the
        probability of the data given the model. The higher the log likelihood, the
        better the model.")
    console.log("")
    console.log("The current log likelihood is:")
    cat("loglik:\n")
    print(q[1])
    console.log("")

    if (waiting) {
      invisible(readline(prompt = "Press [enter] to continue"))
      console.log("")
    }
  }
  while (abs(diff(q[1:2])) >= 1e-6 && it < max_iter) {
    # E step
    comp <- sapply(
      seq_len(k),
      function(i) lambda[i] * dmnorm(data, mu[i, ], as.matrix(sigma[i, , ]))
    )
    comp_sum <- rowSums_finite(comp)
    p <- comp / comp_sum

    if (details) {
      hline()
      console.log(paste0("STEP #", it, ":"))
      console.log("")
      console.log("E-STEP:")
      console.log("")
      console.log("The expectation of each observation to belong to each component of
        the GMM is the following:")
      cat("Expectation:\n")
      print(round(p, 3))
      console.log("")

      if (waiting) {
        invisible(readline(prompt = "Press [enter] to continue"))
```

```
134           console.log("")
135         }
136       }
137
138       # M step
139       lambda <- sapply(
140         seq_len(k),
141         function(i) sum_finite(p[, i]) / nrow(data)
142       )
143       for (i in seq_len(k)) {
144         mu[i, ] <- colSums_finite(p[, i] * data) / sum_finite(p[, i])
145       }
146       for (i in seq_len(k)) {
147         tmp <- wtcov_finite(data, wt = p[, i], center = mu[i, ])$cov
148         sigma[i, , ] <- as.matrix(tmp)
149       }
150
151       if (details) {
152         console.log("M-STEP:")
153         console.log("")
154         console.log("The new components are:")
155         for (i in seq_len(k)) {
156           console.log(paste0("*** Component #", i, " ***\n"))
157           cat("mu:\n")
158           print(mu[i, ])
159           cat("sigma:\n")
160           print(as.matrix(sigma[i, , ]))
161           cat("lambda:\n")
162           print(lambda[i])
163           console.log("")
164         }
165
166         if (waiting) {
167           invisible(readline(prompt = "Press [enter] to continue"))
168           console.log("")
169         }
170       }
171
172       # Compute new expected value of the log likelihood
173       q <- c(sum(log(comp_sum)), q)
174       it <- it + 1
175
176       if (details) {
177         console.log("The new log likelihood is:")
178         cat("loglik:\n")
179         print(q[1])
180         console.log("")
181
182         if (waiting) {
183           invisible(readline(prompt = "Press [enter] to continue"))
184           console.log("")
185         }
186       }
187     }
188
189     comp <- sapply(
190       seq_len(k),
191       function(i) lambda[i] * dmnorm(data, mu[i, ], as.matrix(sigma[i, , ]))
192     )
193     cluster <- apply(comp, 1, which.max)
194     size <- as.integer(table(factor(cluster, levels = seq_len(k))))
```

```
195
196   if (details) {
197     hline()
198     console.log("FINAL RESULTS:")
199     console.log("")
200     if (it == max_iter)
201       console.log("The algorithm stopped because the maximum number of iterations was
202         reached.")
203     else
204       console.log("The algorithm stopped because the change in the log likelihood was
205         smaller than 1e-6.")
206     console.log("")
207     console.log("With the current GMM every observation is assigned to the cluster it
208         is most likely to belong to. The final clusters are:")
209     cat("Cluster assignments:\n")
210     print(cluster)
211     console.log("")
212
213     hline()
214     console.log("")
215   }
216
217   structure(
218     list(
219       cluster = cluster,
220       mu = mu,
221       sigma = sigma,
222       lambda = lambda,
223       loglik = q[2],
224       all.loglik = rev(q[-1])[-1],
225       iter = it,
226       size = size
227     ),
228     class = "gaussian_mixture"
229   )
230 }
231
232 dmnorm <- function(x, mu, sigma) {
233   k <- ncol(sigma)
234
235   x  <- as.matrix(x)
236   diff <- t(t(x) - mu)
237
238   num <- exp(-1 / 2 * diag(diff %*% solve(sigma) %*% t(diff)))
239   den <- sqrt(((2 * pi)^k) * det(sigma))
240   num / den
241 }
242
243 # Finite versions of sum, rowSums, colSums and cov.wt i.e. versions that
244 # replace NA, NaN and Inf values with 1e-300 (ignoring them would lead to
245 # numerical errors)
246 sum_finite <- function(x) {
247   x[!is.finite(x)] <- 1e-300
248   sum(x)
249 }
250
251 rowSums_finite <- function(x) {
252   x[!is.finite(x)] <- 1e-300
253   rowSums(x)
254 }
255
```

```r
colSums_finite <- function(x) {
  x[!is.finite(x)] <- 1e-300
  colSums(x)
}

wtcov_finite <- function(x, wt, center) {
  wt[!is.finite(wt)] <- 1e-300
  cov.wt(x, wt = wt, center = center)
}
```

Code A.3: R implementation of the GMM with EM algorithm

A.4 Agglomerative Hierarchical Clustering

```r
agglomerative_clustering <- function(
  data,
  proximity = "single",
  details = FALSE,
  waiting = TRUE,
  ...
) {
  # Function needed to calculate the avg distance between two clusters
  avg <- function(m1, m2) function(d1, d2) (d1 * m1 + d2 * m2) / (m1 + m2)

  # Figure out the proximity definition
  proximity <- grep(
    tolower(proximity),
    c("single", "complete", "average"),
    value = TRUE,
    fixed = TRUE
  )

  # Exactly one proximity definition should be found
  if (length(proximity) != 1)
    stop("Invalid proximity method")

  if (details) {
    hline()
    console.log("EXPLANATION:")
    console.log("")
    console.log("The Agglomerative Hierarchical Clustering algorithm defines a clustering
        hierarchy for a dataset following a `n` step process, which repeats until a single
        cluster remains:")
    console.log("")
    console.log("    1. Initially, each object is assigned to its own cluster. The matrix
        of distances between clusters is computed.")
    console.log("    2. The two clusters with closest proximity will be joined together
        and the proximity matrix updated. This is done according to the specified proximity.
        This step is repeated until a single cluster remains.")
    console.log("")
    console.log("The definitions of proximity considered by this function are:")
    console.log("")
    console.log("    1. `single`. Defines the proximity between two clusters as the distance
        between the closest objects among the two clusters. It produces clusters where each
        object is closest to at least one other object in the same cluster. It is known as
        SLINK, single-link or minimum-link.")
    console.log("    2. `complete`. Defines the proximity between two clusters as the
        distance between the furthest objects among the two clusters. It is known as CLINK,
```

```
45          complete-link or maximum-link.")
46      console.log("    3. `average`. Defines the proximity between two clusters as the average
47          distance between every pair of objects, one from each cluster. It is also known as
48          UPGMA or average-link.")
49      console.log("")
50
51      if (waiting) {
52        invisible(readline(prompt = "Press [enter] to continue"))
53        console.log("")
54      }
55    }
56
57    # Prepare the data structure which will hold the final answer
58    ans <- structure(
59      list(
60        merge = numeric(0),
61        height = NULL,
62        order = NULL,
63        labels = rownames(data),
64        method = proximity,
65        call = NULL,
66        dist.method = "euclidean"
67      ),
68      class = "hclust"
69    )
70
71    # Create a list with the initial clusters
72    cl <- lapply(
73      seq_len(nrow(data)),
74      function(data) {
75        structure(
76          data,
77          label = -data,
78          members = 1
79        )
80      }
81    )
82    tmp <- sapply(cl, function(x) attr(x, "label"))
83
84    if (details) {
85      hline()
86      console.log("STEP 1:")
87      console.log("")
88      console.log("Initially, each object is assigned to its own cluster. This leaves us with
89          the following clusters:")
90      for (i in seq_len(length(cl))) {
91        cat(paste0(
92          "CLUSTER #", attr(cl[[i]], "label"),
93          " (size: ", attr(cl[[i]], "members"), ")", "\n"
94        ))
95        print(data[cl[[i]], , drop = FALSE])
96      }
97      console.log("")
98
99      if (waiting) {
100       invisible(readline(prompt = "Press [enter] to continue"))
101       console.log("")
102     }
103   }
104
105   # Compute the distances between each point
```

```
106     d <- as.matrix(proxy::dist(data, ...))
107     d <- mapply(
108       "[<-",
109       data.frame(d),
110       seq_len(nrow(data)),
111       sample(Inf, nrow(data), TRUE),
112       USE.NAMES = FALSE
113     )
114     method <- attr(d, "method")
115     ans$dist.method <- if (is.null(method)) "Euclidean" else method
116     dimnames(d) <- list(tmp, tmp)
117
118     if (details) {
119       console.log("The matrix of distances between clusters is computed:")
120       cat("Distances:\n")
121       print(as.dist(round(d, 3)))
122       console.log("")
123
124       if (waiting) {
125         invisible(readline(prompt = "Press [enter] to continue"))
126         console.log("")
127       }
128     }
129
130     for (i in seq_len(length(cl) - 1)) {
131       # Look for the minimum distance between two clusters
132       md <- which.min(d) - 1
133       md <- sort(c(md %% nrow(d), md %/% nrow(d)) + 1)
134
135       # Join the clusters into a new one
136       c1 <- cl[[md[1]]]
137       m1 <- attr(c1, "members")
138       c2 <- cl[[md[2]]]
139       m2 <- attr(c2, "members")
140       c3 <- structure(
141         list(c1, c2),
142         label = i,
143         members = m1 + m2
144       )
145
146       if (details) {
147         hline()
148         console.log(paste0("STEP ", i + 1, ":"))
149         console.log("")
150         console.log("The two clusters with closest proximity are identified:")
151         cat("Clusters:\n")
152         cat(paste0("CLUSTER #", attr(c1, "label"), " (size: ", m1, ")", "\n"))
153         cat(paste0("CLUSTER #", attr(c2, "label"), " (size: ", m2, ")", "\n"))
154         cat("Proximity:\n")
155         print(d[md[1], md[2]])
156         console.log("")
157
158         if (waiting) {
159           invisible(readline(prompt = "Press [enter] to continue"))
160           console.log("")
161         }
162       }
163
164       if (details) {
165         console.log("They are merged into a new cluster:")
166         cat(paste0(
```

```
          "CLUSTER #", attr(c3, "label"), " (size: ", attr(c3, "members"), ")
          [", "CLUSTER #", attr(c1, "label"), " + CLUSTER #", attr(c2, "label"), "]\n"
        ))
        console.log("")

        if (waiting) {
          invisible(readline(prompt = "Press [enter] to continue"))
          console.log("")
        }
      }

      # Add the merged clusters to the answer
      ans$merge <- c(ans$merge, attr(c1, "label"), attr(c2, "label"))
      ans$height <- c(ans$height, d[md[1], md[2]])
      cl <- c(cl, list(c3))
      cl <- cl[-md]
      tmp <- sapply(cl, function(x) attr(x, "label"))

      # Recompute the distances (proximity)
      d1 <- d[, md[1]]
      d2 <- d[, md[2]]
      d3 <- mapply(
        switch(
          proximity,
          single = min,
          complete = max,
          average = avg(m1, m2)
        ),
        d1,
        d2
      )
      d3[md] <- Inf
      d <- cbind(d, d3)
      d <- rbind(d, c(d3, Inf))
      d <- d[-md, -md, drop = FALSE]
      dimnames(d) <- list(tmp, tmp)

      if (details) {
        console.log("The proximity matrix is updated. To do so the rows/columns of the merged
          clusters are removed, and the rows/columns of the new cluster are added:")
        cat("Distances:\n")
        print(as.dist(round(d, 3)))
        console.log("")

        if (waiting) {
          invisible(readline(prompt = "Press [enter] to continue"))
          console.log("")
        }
      }
    }

    # Compute the merge and order of the hclust
    ans$merge <- matrix(ans$merge, ncol = 2, byrow = TRUE)
    ans$order <- unlist(cl)

    if (details) {
      hline()
      console.log("RESULTS:")
      console.log("")
      console.log("Since all clusters have been merged together, the final clustering hierarchy is:")
      console.log("(Check the plot for the dendrogram representation of the hierarchy)")
```

```
    plot(ans, hang = -1)
    console.log("")

    if (waiting) {
      invisible(readline(prompt = "Press [enter] to continue"))
      console.log("")
    }

    hline()
  }

  # Return the answer
  ans
}
```

Code A.4: R implementation of the AHC algorithm

A.5 Divisive Hierarchical Clustering

```
divisive_clustering <- function(
  data,
  details = FALSE,
  waiting = TRUE,
  ...
) {
  if (details) {
    hline()
    console.log("EXPLANATION:")
    console.log("")
    console.log("The Divisive Hierarchical Clustering algorithm defines a clustering hierarchy
        for a dataset following a `n` step process, which repeats until `n` clusters remain:")
    console.log("")
    console.log("")
    console.log("    1. Initially, each object is assigned to the same cluster. The sum of squares
        of the distances between objects and their cluster center is computed.")
    console.log("    2. The cluster with the highest Within-Cluster Sum-of-Squares (WCSS) is split
        into two using the K-Means algorithm. This step is repeated until `n` clusters remain.")
    console.log("")
    console.log("Since this implementation builds a complete hierarchy, the second step does
        not need to be performed on the cluster with the highest sum of squares, but rather
        on any cluster with more than one element.")
    console.log("")

    if (waiting) {
      invisible(readline(prompt = "Press [enter] to continue"))
      console.log("")
    }
  }

  # Prepare the data structure which will hold the answer
  ans <- structure(
    list(
      merge = numeric(0),
      height = NULL,
      order = NULL,
      labels = rownames(data),
      method = "kmeans",
      call = NULL,
```

```
      dist.method = "Euclidean"
    ),
    class = "hclust"
  )

  # Wrap the data with additional information we'll need
  data_center <- apply(data, 2, mean)
  totss <- sum(apply(data, 1, function(x) x - data_center)^2)
  wrapped_data <- list(
    data = data,
    label = nrow(data) - 1,
    ss = totss,
    elems = -seq_len(nrow(data))
  )

  # Build a list with all clusters
  clusters <- list(wrapped_data)

  if (details) {
    hline()
    console.log("STEP 1:")
    console.log("")
    console.log("Initially, each object is assigned to the same cluster. The sum of squares
        of the distances between objects and their cluster center is computed.")
    cat("Initial cluster:\n")
    cat(paste0("CLUSTER #", clusters[[1]]$label, ":\n"))
    print(clusters[[1]]$data)
    console.log("")

    if (waiting) {
      invisible(readline(prompt = "Press [enter] to continue"))
      console.log("")
    }
  }

  # Until there are no clusters with sum of squares greater than 0
  iter <- 2
  while (any(sapply(clusters, function(x) length(x$elems)) > 1)) {
    # We'll operate on the first cluster and order them all afterwards
    target <- clusters[[1]]
    clusters <- clusters[-1]

    if (details) {
      hline()
      console.log(paste0("STEP ", iter, ":"))
      console.log("")
      console.log("Any cluster is selected for division:")
      cat("Cluster:\n")
      cat(paste0("CLUSTER #", target$label, " (WCSS: ", target$ss, ")\n"))
      console.log("")

      if (waiting) {
        invisible(readline(prompt = "Press [enter] to continue"))
        console.log("")
      }
    }

    # Split the target cluster into two using the k-means approach
    km <- clustlearn::kmeans(target$data, 2, ...)
    if (any(km$size == 0))
      km$cluster[1] <- setdiff(1:2, km$cluster[1])
```

```
101
102       # Create clusters for each split
103       lhs <- list(
104         data = target$data[km$cluster == 1, , drop = FALSE],
105         label = NULL,
106         ss = km$withinss[1],
107         elems = target$elems[km$cluster == 1]
108       )
109       lhs$label <- if (length(lhs$elems) == 1) {
110         lhs$elems
111       } else {
112         target$label - 1
113       }
114       rhs <- list(
115         data = target$data[km$cluster == 2, , drop = FALSE],
116         label = NULL,
117         ss = km$withinss[2],
118         elems = target$elems[km$cluster == 2]
119       )
120       rhs$label <- if (length(rhs$elems) == 1) {
121         rhs$elems
122       } else {
123         target$label - length(lhs$elems)
124       }
125
126       if (details) {
127         console.log("The cluster is divided through the K-Means method into two that
128           (approximately) minimize the WCSS:")
129         cat(paste0("CLUSTER #", lhs$label, " (WCSS: ", lhs$ss, ")\n"))
130         print(lhs$data)
131         console.log("")
132         cat(paste0("CLUSTER #", rhs$label, " (WCSS: ", rhs$ss, ")\n"))
133         print(rhs$data)
134         console.log("")
135
136         if (waiting) {
137           invisible(readline(prompt = "Press [enter] to continue"))
138           console.log("")
139         }
140       }
141
142       # Update the answer
143       ans$merge <- c(lhs$label, rhs$label, ans$merge)
144       ans$height <- c(km$totss, ans$height)
145
146       if (details) {
147         console.log("If the divided clusters containe more than one element, they are
148           marked again for division:")
149       }
150
151       # Replace the target cluster with the two new ones
152       if (length(rhs$elems) > 1) {
153         clusters <- c(list(rhs), clusters)
154
155         if (details) {
156           console.log("The following cluster is marked for division:")
157           cat(paste0("CLUSTER #", lhs$label, " (WCSS: ", lhs$ss, ")\n"))
158           print(lhs$data)
159         }
160       }
161       if (length(lhs$elems) > 1) {
```

```
162         clusters <- c(list(lhs), clusters)
163         if (details) {
164           console.log("The following cluster is marked for division:")
165           cat(paste0("CLUSTER #", rhs$label, " (WCSS: ", rhs$ss, ")\n"))
166         }
167       }
168
169       if (details) {
170         console.log("")
171
172         if (waiting) {
173           invisible(readline(prompt = "Press [enter] to continue"))
174           console.log("")
175         }
176       }
177
178       iter <- iter + 1
179     }
180
181     # Compute the merge and order of the hclust
182     ans$merge <- matrix(ans$merge, ncol = 2, byrow = TRUE)
183     ans$height <- sqrt(ans$height)
184
185     # Order the rows in merge by height
186     tmp <- order_merge_by_height(ans$merge, ans$height)
187     ans$merge <- tmp$merge
188     ans$height <- tmp$height
189
190     # Figure the order of the elements from the merge to build a proper dendrogram
191     ans$order <- merge2order(ans$merge)
192
193     if (details) {
194       hline()
195       console.log("RESULTS:")
196       console.log("")
197       console.log("Since all clusters have been divided into clusters with a single element,
198           the final clustering hierarchy is:")
199       console.log("(Check the plot for the dendrogram representation of the hierarchy)")
200       plot(ans, hang = -1)
201       console.log("")
202
203       if (waiting) {
204         invisible(readline(prompt = "Press [enter] to continue"))
205         console.log("")
206       }
207
208       hline()
209     }
210
211     # Return the answer
212     ans
213   }
214
215   order_merge_by_height <- function(merge, height) {
216     ord1 <- order(height)
217     ord2 <- order(ord1)
218     for (i in seq_len(nrow(merge))) {
219       for (j in seq_len(ncol(merge))) {
220         val <- merge[i, j]
221         merge[i, j] <- if (val < 0) val else ord2[val]
222       }
```

```
223     }
224     merge <- merge[ord1, , drop = FALSE]
225     height <- height[ord1]
226 
227     list(merge = merge, height = height)
228   }
229 
230   merge2order <- function(merge) {
231     order <- if (nrow(merge) > 0) merge[nrow(merge), ] else -1
232     while (any(order > 0)) {
233       target <- which.max(order)
234       order <- c(
235         order[seq_along(order) < target],
236         merge[order[target], ],
237         order[seq_along(order) > target]
238       )
239     }
240     abs(order)
241   }
```

Code A.5: R implementation of the DHC algorithm

A.6 Auxiliary functions

```
1   console.log <- function(txt, ...) {
2     width <- console_width()
3 
4     # Split the text into lines
5     tmp <- strsplit(txt, '[\r\n]')[[1]]
6 
7     # Process every line
8     lines <- NULL
9     for (line in tmp) {
10      # Get the whitespaces at the beginning of the line
11      white <- get_whitespace(line)
12 
13      # Remove trailing whitespace from the line
14      line <- substring(line, nchar(white) + 1)
15      if (is.na(line)) line <- ''
16 
17      # Expand the tabs in the whitespace
18      white <- gsub('\t', '  ', white)
19 
20      # Split the line into several lines which will fit in the console
21      parts <- strsplit(
22        line,
23        paste0("(?<=.{", max(width - nchar(white), 1), "})"),
24        perl = TRUE
25      )[[1]]
26 
27      # Add the whitespace to the beginning of each part
28      parts <- paste0(white, parts)
29 
30      # Add all parts to the processed lines list
31      lines <- c(lines, parts)
32    }
33 
34    # Make sure to print something...
```

```
35    if (length(lines) < 1) lines <- ""
36
37    # Print all processed lines
38    for (line in lines) {
39      message(line, ...)
40    }
41  }
42
43  hline <- function() {
44    console.log(strrep('_', cli::console_width()))
45  }
46
47  get_whitespace <- function(txt) {
48    # Find whitespace at the beggining of the string
49    fst_match <- gregexpr("^[ \t]*", txt)[[1]]
50
51    # Extract whitespace
52    white.length <- attr(fst_match, "match.length")
53    substring(txt, 1, white.length)
54  }
```

Code A.6: Auxiliary functions to log the explanations

Universidad de Alcalá
Escuela Politécnica Superior

ESCUELA POLITECNICA
SUPERIOR