POLITECNICO DI MILANO

Scuola di Ingegneria dei Sistemi
Corso di Laurea Magistrale in Ingegneria Matematica

Big Data improvements in cluster analysis

Relatore: Dott.ssa Ilenia Epifani
Correlatore: Dott. Gianfranco Calvi

Tesi di Laurea di:
Michele Usuelli
770503

Anno Accademico 2011-2012

Contents

List of Figures

Abstract

Negli ultimi anni, si sta parlando sempre più spesso di "Big Data", riferendosi non solo a grandi moli di dati. Infatti, l'espressione riguarda alcune nuove necessità e le conseguenti sfide, dette le "Tre V": Volume, cioè gestione di grandi moli; Velocità, cioè rapidità di analisi; Varietà, cioè elaborazione di dati non strutturati, come testi, immagini e video.

Questa tesi tratta l'utilizzo di tecniche di clustering in questo nuovo contesto. Il clustering consiste nella segmentazione di un insieme di oggetti in gruppi che siano il più possibile omogenei. Di fronte a grandi moli di dati, il clustering è uno strumento potente che produce un piccolo insieme di gruppi, facilmente trattabile. Inoltre, le tecniche presentate sono particolarmente efficienti, quindi uno strumento di calcolo adeguato risolve il problema della velocità. Per quanto riguarda la varietà, esistono strumenti di clustering che trattano anche dati non strutturati, ma non sono parte della tesi.

Il software che è stato scelto è "Hadoop", molto utilizzato in ambito "Big Data" in quanto permette di gestire grandi moli di dati con un costo contenuto e di trattare dati non strutturati. Esso è basato sulla gestione di grandi volumi mediante la distribuzione del lavoro su un cluster di computer. A tal fine, gli algoritmi sono stati sviluppati in uno specifico paradigma, detto "MapReduce", che consente la loro parallelizzazione mediante Hadoop. Per questo motivo, alcuni algoritmi di clustering già esistenti sono stati adattati alla struttura del MapReduce e altri sono stati sviluppati direttamente seguendo questa logica.

Il lavoro di tesi è consisito nello sviluppo di alcuni algoritmi, che sono stati poi testati su dataset simulati. La prima fase di testing ha riguardato l'efficacia degli algoritmi, cioè la loro capacità di segmentare correttamente un insieme di oggetti. A tal fine, sono stati utilizzati dataset di piccole dimensioni e aventi caratteristiche particolari. L'altra fase ha riguardato l'efficienza, cioè la rapidità di esecuzione, e il testing è stato condotto su dataset di dimensioni maggiori tramite un cluster Amazon di 5 nodi. Nonostante il volume dei dati trattati sia ancora relativamente piccolo, è possibile stimare le prestazioni su moli maggiori. Infatti, il MapReduce ha la peculiarità di essere scalabile. Questo significa che la potenza di calcolo cresce linearmente all'aumentare delle risorse, quindi è sufficiente aumentare il numero di nodi in proporzione alla mole di dati da processare per ottenere le stesse prestazioni.

In conclusione, la parte innovativa del lavoro di tesi consiste nella progettazione e implementazione di algoritmi di clustering in MapReduce. Essi sono

basati sulla combinazione di logiche di algoritmi già esistenti, riadattate nel nuovo paradigma.

Abstract

The expression "Big Data" has become very popular in the last few years though it does not concern exclusively large volumes of data. In fact, it is more connected to the way the data needs to be treated and the consequential challenges, called the "Three V", that stand for Volume (treatment of large datasets) , Velocity (quickness of analysis), and Variety (handle of unstructured data, such as texts, images, and videos).

This paper discusses the use of clustering in this context, though not considering the challenge of variety. Clustering consists in the segmentation of a set of objects through the identification of features, so to group them accordingly as much as possible. By all means, this approach reduces the size of the problems involved, as a wide dataset can be handled as though as it were a small set of clusters. Furthermore, the paper describes efficient algorithms, which can be used to create the appropriate tools to allow quick data processes, thereby dealing effectively with the velocity challenge.

For this purpose, the choice of software framework went for Hadoop, as it allows a cheap processing of large volumes of data and the handling of unstructured data. The logic upon which it is based is the parallelization of processes using a cluster of computers. For this purpose, the clustering algorithms have been developed through a specific programming model, i.e. MapReduce, since it allows the parallelization of tasks. Therefore, some of the current clustering algorithms have been converted to the MapReduce structure, while others have been developed straight away in that manor.

Once the tools were designed, the testing was conducted on some simulated datasets. The first stage regarded the effectiveness, i.e. the capability of identifying correctly some unusually shaped clusters. Therefore, the used dataset were small-sized. Consequently, the efficiency testing aimed to cluster the big dataset more rapidly. For this stage, the used tool was an Amazon cluster of 5 computers.

Although the tested volume was still pretty small, it is possible to estimate the performance changes as the dataset grow. As a matter of fact, one of the MapReduce peculiarities is its scalability, i.e. the capability to increase linearly the computational power as the resources grow. Hence, if the size of the cluster is proportional to the data volume, the performances are approximately constant.

In conclusion, the design and the development of these new clustering algorithms in MapReduce combines the logics of two current classes of clustering

algorithms. By all means, this approach has the advantages of both and gives, therefore, a new range of efficient analytical methodologies and consequential results.

Chapter 1

Introduction

In the last years, digital technologies, social networks, and forums have been propagating much more than in the past decades. Moreover, their growth is expected to quicken further. As regards the most of private companies, data volume is much smaller, but it is growing a lot too. These are just a few examples of the data deluge affecting almost everything.

It is possible to extract useful information from them through the techniques of data analysis. The target is the solution of a wide set of problems, such as interaction with customers, strategic decisions, and processes optimization. For instance, many chains of stores track their customers' purchases using data extracted from their fidelity cards. In addition, the integration between data coming from companies and from the web can produce further information. In this way, some companies try to understand the opinions about their products by analyzing forum discussions. These examples are only a small part of the multitude of results that can be extracted from the huge mine of data.

Regarding current approaches, such as the use of databases, the starting point is often the selection of small datasets from the whole, including only information that is highly relevant to the problem, on the base of analysts' opinions. Since most of the tools do not allow a cheap treatment of wider sets of data, this is the most common way to proceed. However, this approach has some disadvantages since it is not always possible to determine with enough precision which information is useful and which can be excluded. Maybe it was not a considerable problem when the overall data were small, but the data spread has worsened the situation. Moreover, the introduction of less relevant data in analysis can lead to more precise results and to a bird's eye understanding of

the problem, so the ideal tool exploits all the available information.

There are some tools that can be integrated in the current devices, in order to handle wider datasets, but these solutions are often afterthoughts and they imply some efficiency problems. In fact, the cost often becomes much higher. Furthermore, the maximum amount of data analyzable is not growing as fast as data.

1.1 New challenges

The target is to extend the analysis to all available information, so it is necessary to use new techniques. According to some companies [1] [2], such as IBM, the new challenges are the “Three V" that stand for Volume, Velocity, and Variety. “Big Data" phrase refers to all data analysis that present any of these new necessities.

1.1.1 Volume

According to some IBM estimations, the amount of data produced every day is about 2.5 quintillion of bytes and the most of it (about 90%) belongs to the last two years. Furthermore, according to the current trend, the growth is speeding up, so it is just the beginning. This fact is just an index of how recent the volume problem is. In regards to the total size of data, in 2009 it was about 0.8 zettabytes, i.e. 0.8×10^{21} bytes, and, according to the trend, in 2020 it will be 44 times more [4]. Figure 1.1 shows a trend estimation.

The main data source is the Internet since the web diffusion affects billions of people and each one of them produces a lot of data. Indeed, almost every forum, social network, and chat gathers the thoughts of thousands or even millions of people. Just to give some numbers, Facebook generates every day about 10 terabytes of data and Twitter about 7 terabytes. These data can be used to extract information about many situations. For instance, it is possible to evaluate the opinions about some products, assuming that they will have a significant impact on the sales.

As regards most of private companies, an example of a data source is given by electronic devices, such as Smartphones and sensors. In fact they provide track of the activities, so they contain a lot of useful information. For example, it is possible to analyze car movements, using data coming from GPS, in order to manage traffic.

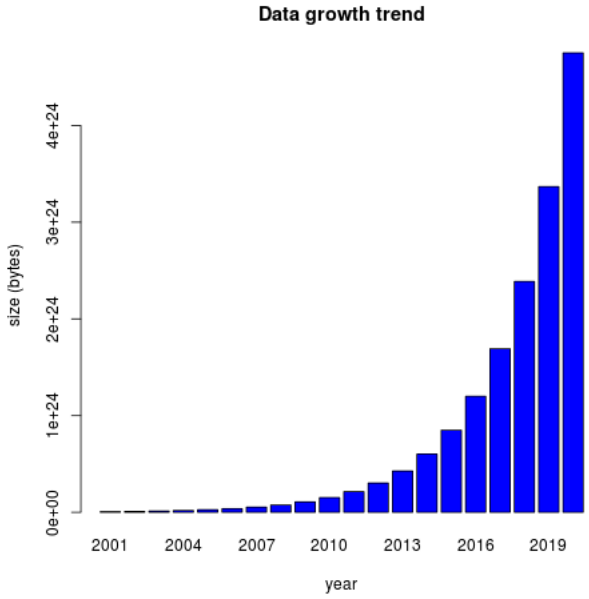

Figure 1.1: Data growth trend estimation.

Of course not all data will be used, but, combining stored data with web information and devices tracks, a single problem can be related with terabytes of even petabytes of useful data. Clearly, even if the analysis does not consider the web, the volume can be very big. Unfortunately, current technologies imply an elevated cost for the treatment of big volumes. For this reason, there is a need for a new kind of data storage and processing.

1.1.2 Velocity

Nowadays, the world is a lot quick-change, so it is often necessary to take fast decisions. The amount of data that are generated every day is very big and it is necessary to gather, store, and analyze them in a fast way. Sometimes it is even necessary to have a real-time track of facts that comes from the collection and analysis of streams of data coming from different sources. Some examples of problems that need this kind of information are financial investments planning, weather forecast, and traffic management.

In order to speed up an analysis, the easiest way is to store data in a fast

and efficient memory, i.e. the RAM of the computer. This approach, called in-memory processing, is now used a lot. For example, the statistical software R is mostly based on that. Unfortunately, fast and efficient memory devices are a very expensive, so their size is limited. For that reason, the RAM of computers can contain only a few gigabytes. Clearly, this approach does not work well with Big Data problems, so it is necessary to find an alternative.

As regards on-database storage, analysis is much slower, and, as data size grows, the computational cost explodes. Consequently, the solution is a tool that allows a fast access and process of big amounts of data, by using a cluster of computers, as described subsequently.

1.1.3 Variety

A few years ago, analysis used to deal only with well-structured sets of data. However, most of data growth is due to websites as they contain texts, images, videos, and log files. Each of these kinds of data is structured in its own way, so it requires a proper handle. Furthermore, also data generated by sensors are usually unstructured. Of course it is not ways effective to ignore all these useful information, so new tools must be able to treat all kinds of data, or at least the most of them.

A possibility is to use an approach that is partly equal to the current one. It consists in the structuring data with new tools, before conducting an analysis through classical tools. For example, data scraping is a technique that translates human-readable information, such as texts, in computer language, i.e. in structured data.

A better way is to develop new techniques that directly analyze all data. Since there are different kinds of unstructured data, each one of them needs a proper tool. Therefore, this approach leads to some complications and to the use of a wider number of new techniques.

1.2 Big Data improvements on clustering

Section 1.1 gave an overview of the wide Big Data world, but this paper is only about a small part of the techniques. In fact, it deals with cluster analysis in a Big Data environment. The aim of this section is to give an overview of the issues and of the techniques.

1.2.1 Cluster analysis overview

The target of data analysis is to obtain a concise information about a set of objects. If the number of them is too high and the analysis considers each one of them individually, the information is too detailed, so it is impossible to have useful results. For this reason, the first step consists in a reduction of the size of the problem. The approach that we present consists in dividing the objects in clusters, so that the ones that belong to the same are similar. In this way, the clusters are as homogeneous as possible and it becomes enough to show results about each one of them. Afterwards, each datum is automatically affected by results about the cluster it belongs to.

The aim of clustering is to identify the clusters, and to assign each object to the right one. In order to combine the objects and to determine which ones are joined in the same cluster, it is necessary to use a criterion that evaluates a similarity between them. A possible approach is to compute the distances between the objects, i.e. values that express the difference between them. If they are described by features, the distance simply represents the dissimilarity between their values. However, there are also other possibilities. For example, if there is an interaction between the objects, it is possible to build a network that plays a role in the distances computation.

In addition to the distance criterion choice, the clustering mechanism depends on the chosen algorithm. There are many main classes of methods and the two most important are the following.

Hierarchical clustering First association of near objects, then choice of the clusters.

Centroid-based clustering First clusters identification, then assignment of each object to the nearest cluster.

1.2.2 Why Big Data in clustering?

For many years, cluster analysis has been conducted using traditional approaches. However, it is one of the best and most evident example of a situation that is affected by all Big Data problems. In fact, cluster analysis is afflicted by the "Three V" mentioned in Section 1.1.

An analysis can now treat even millions or billions objects. In those situations, clustering is particularly important since it simplifies the problem by generating a few clusters from the multitude of objects. Not all the current

clustering techniques are capable of handling large volumes, so some of them cannot be used at all.

Even if an algorithm is adaptable to the handle of big volumes, its computational cost often explodes in those situations and the analysis needs an answer in a reasonable time. Furthermore, nowadays many problems require an even quicker data process. For these reasons, there is a need of more efficient algorithms.

Another problem of cluster analysis is the variety handle. In fact, in some situations, unstructured data contain much information that should be considered by the analysis. Some examples of unstructured information are networks that link objects and texts to compare. The easiest approach is to convert that information in structured data, processed through a traditional algorithm. By the way, a better result is obtained if the clustering algorithm can directly handle the variety. An example of these techniques is cluster analysis of graphs.

1.3 Thesis organization

The target of this paper is to present some techniques that allow to perform cluster analysis on big volumes of data. Although clustering is affected by all the “Three V" mentioned in Section 1.1, this paper deals only with the volume and velocity challenges.

Chapter 2 describes the Hadoop framework that allows the use of the techniques, in order to show the logic upon which it is based. In fact, Hadoop has some particular requirements regarding the algorithms structure. The advantage of its use is the scalability of the techniques, i.e. their capability to increase linearly the computing power as the resources grow. This fact is allowed by the use of a specific programming model for the algorithms that is MapReduce. Although this is not the only possibility, Hadoop fits the problem treated in this paper.

Chapter 3 describes some clustering algorithms. The easiest approach is to adapt some current clustering techniques to the new environment. In detail, the reference algorithm is K-means since it is efficient and easily adaptable into Big Data problems. An other option, treated in Chapter 4, is the use of new algorithms that fulfill the MapReduce requirements. The solution consists in the combination of two kinds of techniques, in order to have, as much as possible, the advantages of both. The algorithms have been developed in R and tested

on some datasets. The code is described in Appendix B.

Once the current algorithms have been chosen and the new have been designed, the work has consisted in their implementation in MapReduce. Chapter 5 describes the steps and Appendix A shows the code. The chosen software is R, connected with Hadoop. The advantages of using R, rather than some others programming languages, is the possibility of integrating the overall analysis process in the same environment. In fact, it allows both the MapReduce implementation and the analysis of results.

Finally, in order to evaluate the computational cost of the algorithms, they have been tested on different-sized datasets. Chapter 6 presents the results and Chapter 7 discusses the overall work.

Chapter 2

Hadoop framework

This chapter describes a possible solution for the handle of some Big Data problems that consists in the use of Hadoop [5]. It is an open-source framework that is written in Java and it allows the handle of big volumes of data through a scalable and distributed system. Hadoop runs on commodity hardware, i.e. not expensive and also available on demand on the cloud, so the main advantage that derives from it is the reduction of the cost of data storage and process, particularly when the volume increases. Furthermore, it allows the handle of unstructured data, so it can solve a wide range of problems when compared with more traditional approaches.

2.1 Storage of big volumes of data

The traditional approach is to keep data in structured repositories, such as RDBMS (Relational DataBase Management System). An advantage that derives from that is a better organization of information since data are directly structured during the loading. However, the use of this approach to handle large volumes of data leads to a cost explosion. Furthermore, relational databases treat well homogeneous data, but they do not allow the handle of unstructured data, so they are not a good solution in some situations.

As mentioned in Subsections 1.1.1 and 1.1.3, the analysis may require the use of big and unstructured volumes of data. This section describes the Hadoop solution to the volume problem that is based on the use of cheap devices.

2.1.1 Distributed file system: HDFS

One of the advantages of Hadoop is the scalability, i.e. the ability to handle a growing amount of work with the same level of performance. This means that, as the available resources grow, the computational capacity grows in a linear way. In this case, the maximum allowed storage is proportional to the number of computers in the cluster, so the volume problem is handled by increasing the size of the cluster. The cheapness comes from the fact that it is not necessary to make use of expensive devices, but it is sufficient to increase the number of cheap computers.

Before processing the data, Hadoop stores them through HDFS[6] (Hadoop Distributed File System). In fact, this kind of storage is thought to allow a fast and scalable data process.

Regarding of the structure, data are stored as key-value pairs. It means that each datum is constituted by a value that represents the datum itself, associated with a key that identifies it and allows its access. As regards of the shape of the value, it has not got any particular requirement, so it is not necessary to treat data before they are stored. In fact, the task of treating in proper ways different kind of data is left to the analysis.

Another advantage of HDFS is its fault-tolerance. In fact, the data are stored in multiple copies, so the failure of a single computer of the cluster does not have any relevant consequences. Due to this fact, Hadoop does not have any particular requirement about the hardware. In fact, it can use without any problems commodity hardware, i.e. common available hardware that is cheap and easily affectable by failures.

As regards the data import, there is a master node, i.e. a chosen computer that gradually receives data, splits them in blocks and scatters them around the cluster. The block size is very big, if compared to the one of other file systems, in order to improve the efficiency. In detail, the default block size is about 64 MB or 128 MB, due to the results of the following calculation. Let t_r be the transfer rate and let s_t be the seek time. If the purpose is to have $t_r = 0.01\, s_t$ and the hardware has $t_r = 100\, MB/s$ and $s_t = 10\, ms$, then the ideal chunk size is $100\, MB$.

2.2 Algorithms in the Hadoop environment

Similarly to the data storage, the data analysis is conducted through a proper scalable technique. This section presents the programming model provided by Hadoop.

2.2.1 MapReduce paradigm

In order to process data, Hadoop uses a programming model developed by Google, named "MapReduce" [3]. Due to its scalability, it allows the process of huge amounts of data that have been stored through HDFS.

The process management is based upon a hierarchy between the computers in the cluster. In detail, there is a master node that coordinates the jobs by scattering the operations through the other computers, called workers. Moreover, this hierarchy can be built on different levels.

The scalability derives from the logic upon which the job is split. In fact, the dataset is divided into chunks, each one of which is assigned to a worker. Then, the master task is only to coordinate the jobs, so its workload is only a small part of the total and all processes are committed to the workers.

As its name suggests, MapReduce is based upon the separation of algorithms in the following two steps that will be described in detail afterwards.

Map Separate process of dataset chunks.

Reduce Collection of the Map outputs.

The input is made off by HDFS data that are stored in the form of key-value pairs. Moreover, the algorithm requires the storage of intermediate files during the computation of the output and, in order to handle them, these files are stored as key-value pairs too.

In order to explain the paradigm, let X be the dataset and let $\{X_1, ..., X_N\}$ be its partition, in such a way that each chunk X_i is associated with a task. For simplicity, since now, let us assume that each node always performs one task. Accordingly, the implementation consists in the following steps.

Input split The master receives the job and it splits it in small tasks, each one of which analyzes a single dataset chunk. $\forall i = 1, ..., N, X_i$ is assigned to a worker that will process it.

Map For each task, the corresponding worker maps its chunk, producing a key/value output. Mapping consists in applying a function that is the same for all the subsets.

$$Map(X_i) = (k_i, V_i) = \{(k_i^j, V_i^j)\}_{j=1,...,J_i}$$

where k_i is the vector of keys, V_i is the matrix containing the corresponding values, and J_i is the output size.

Shuffle Data with the same key are grouped and moved into the same place. In formulas, let K be the set of all possible values of the keys. Then, $W_k = \{V_i^j : k_i^j = k, i = 1, ..., N\}, \forall k \in K$.

Reduce For any key, the master[1] computes a result by applying the reduce function to the set of data associated with it. $Y_k = Red(W_k), \forall k \in K$.

Output collection Output coming from the Reduce processes are collected, producing the results. $Y = \{Y_k\}_{k \in K}$.

Example 2.1 illustrates the logic upon which MapReduce is based.

***Example* 2.1** (Wordcount)**.** *The aim is to count the number of occurrences of each word in a book. The result is a list of words, associated each with the number of its occurrences. In order to parallelize the jobs, the algorithm splits the text in chapters and it assigns each one of them to a worker. The steps are the following.*

Input split *The master assigns to each worker the task of mapping a single chapter.*

Map *Since the output will be a list of words, the worker maps each one of them with its key that is the word itself. In order to count the number of occurrences, the value is their number that in this case is always equal to "1". In fact, each word is present just once when is mapped.*

Shuffle *For any word, all data are put in the same place, in order to process them.*

[1] In the practice, any value of the key is associated with a task that can be executed by the master or by a worker.

Reduce *The master computes the sum of occurrences of any word. It means that it counts the number of times the word is present in the whole of mapping outputs.*

Output collection *The master produces a list of all words, associated with their count.*

Figure 2.1 shows this MapReduce example.

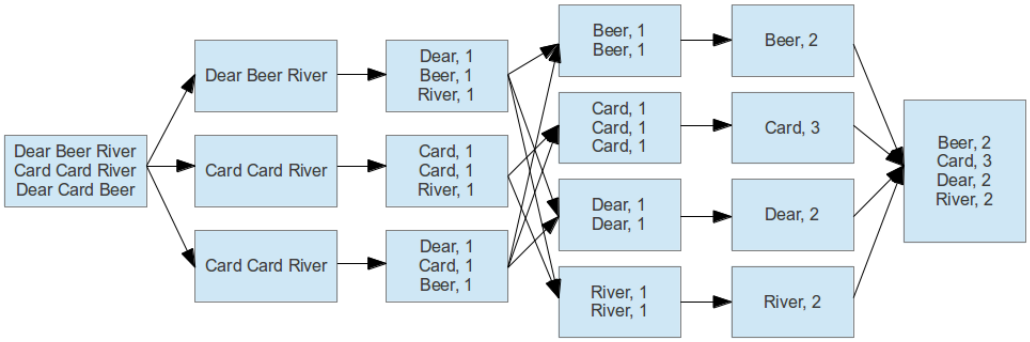

Figure 2.1: An example of MapReduce: wordcount

Clearly, not all algorithms can be parallelized via MapReduce. In fact, there must be the appropriate conditions to set them as parallelizable. Furthermore, the splitting is not automatic, so Map and Reduce functions must be chosen ad hoc.

2.2.2 Combiner use

If the Map outputs are too big, the process of all data that are mapped with the same key can be problematic. In fact, it leads to a bottleneck that slows down the entire process. A possible solution is to split the Reduce operations by introducing another step.

After the Map step, each worker can apply a function to its data, in order to reduce the size of its output. This step is called Combine, expressing the fact that the workers combine data before sending them to the master, and it is used

before the shuffle step. It is not always possible, but, under some conditions, it can even be a function identical to the Reduce one. In other situations, instead, it is not the same function. After the Combine step, the master collects the outputs of all nodes by applying the Reduce function.

Input split The master receives the job and it splits it in small tasks, each one of which analyzes a single dataset chunk.

Map For each task, the corresponding worker maps its chunk, producing a key/value output.

Combine Within any chunk, the corresponding worker applies the Combine function to the data belonging to the corresponding dataset and mapped with the same key.

Shuffle Data associated with the same key are grouped and moved to the same place

Reduce For any key, the master computes a result by applying the Reduce function to the set of data associated with it.

Output collection The outputs that are generated by Reduce processes are collected, producing the result.

The addition of Combine is a good choice, as it decreases the computational cost, but when is it possible? As mentioned before, in some cases it is sufficient to apply the Reduce function one more time. In order to allow this, the result with its use must be identical to the Combine-less MapReduce. Hence, calling $W_k^i = \{V_i^j : k_i^j = k\}$, it must be true that $Red(\{Red(W_k^i)\}) = Red(\bigcup_{i=1,\dots,N} W_k^i)$, i.e. that the Reduce function must be associative. Furthermore, since nodes are not ordered, function must be commutative too.

Another possibility is to use a Combine function that is different from the Reduce one. Because of the variety of algorithms, there is not a specific rule, so each case must be treated in a different way.

In the case of word-count, the Reduce function is a sum that is a commutative and associative operation. Hence, MapReduce with Combiner is possible and it consists in the steps described in Example 2.2.

***Example* 2.2** (Wordcount with Combiner)**.** *The aim is to count the number of occurrences of each word in a book. The result will be a list of words, associated*

each with the number of its occurrences. In order to parallelize the job, the master splits the text in chapters and assigns each one of them to a worker.

The algorithm steps are the following.

Input split *The master assigns to each worker the task of mapping a single chapter.*

Map *Since the output will be a list of words, the worker maps each one of them with a key that is the word itself. In order to count the number of occurrences, the value is their number that in this case is always equal to "1". In fact, each word is present just once when it is mapped.*

Combine *Any worker computes the partial wordcount.*

Shuffle *The master lays out all the words.*

Reduce *The master computes the sum of occurrences of any word. It means that it sums up the Combine count outputs.*

Output collection *The master produces a list of all words, associated with its count.*

2.3 A part of the Hadoop ecosystem

Hadoop has some useful components that ease the data handle. Here is reported a list of the most important among them.

Sqoop A connector between Hadoop and many databases.

Hive A tool that allows to use Hadoop as though as it was a data warehouse.

Pig A batch-oriented high-level query language.

Mahout A data-mining and artificial intelligence library.

Flume A log and text files import tool.

Hbase A column-oriented key-value database.

Zookeeper A configuration and synchronization service.

Since the analysis described in this paper does not need all of them, this section describes only the ones that have been used.

2.3.1 Connection with other databases: Sqoop

Hadoop ecosystem deals with different-shaped data and, in most of the situations, some of them are in structured databases. The aim of Sqoop is to integrate this kinds of data into the environment.

Regarding the source, Sqoop supports the connection with several relational databases, such as MySQL and Oracle, with data warehouses, and with NoSQL stores. Starting from them, it extracts data, puts them into Hadoop, and stores them using HDFS filesystem. Furthermore, it also supports HBase and Hive storage. Once data has been analyzed through Hadoop, Sqoop allows also to store the results in the starting relational database.

In order to parallelize the job, the extraction is made through MapReduce. In detail, each map imports a part of the database and stores it into a HDFS chunk, as shown in Figure 2.2.

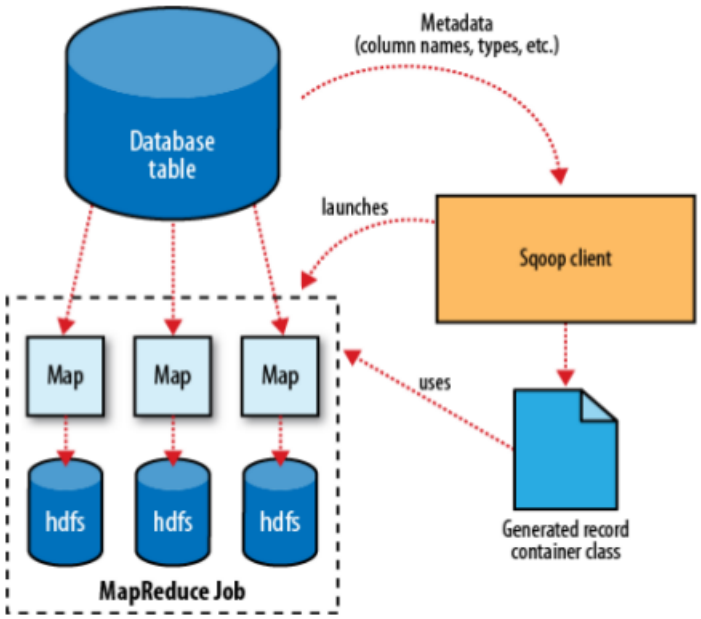

Figure 2.2: Sqoop import process

2.3.2 Hadoop DWH: Hive

Hive [7] is a framework originally developed by a Facebook team and it is a data warehouse system for Hadoop. In particular, using Hive it is possible to handle data through Hive Query Language (HQL) that is much similar to SQL. The logic upon which it is based is to translate SQL-like queries in MapReduce jobs, in order to have a fast and scalable data process.

Nevertheless, although the data process is similar to the one of a traditional database, the use of MapReduce implies some structural differences. The main dissimilarity is about the table schema. In fact, traditional databases use a "schema on write", i.e. the comparison between the data and the schema during the storage. In this way, related tables are already stored near, in order to quicken the analysis. On the other hand, Hive uses a "schema on read". It means that the data are not structured when they are loaded, but when a query is issued. The advantage is that in this way the data load is faster because of the lack of checking. However, the "schema on write" allows to speed up the queries because of the indexing and the compression during the data load. This trade-off is in favor of Hive in situations in which the schema is not already known during the loading.

A Hive disadvantage is the lack of the possibility of updating the table. In fact, MapReduce implies the access of all data at the same time and the storage of data into a new table.

2.4 R with Hadoop

R is a powerful and flexible analysis software. In fact, it provides many useful statistical tools and it allows to develop custom algorithms. However, it mainly deals with in-memory analysis, in contrast with Big Data needs. Even if there are some packages that allow the handle of big volumes of data, they are not as efficient as Hadoop.

Fortunately, there are also some connectors between R and the Hadoop framework. In this way, there are some packages that allow to develop MapReduce algorithms directly from R. As typically an algorithm requires the parallelization of only some steps, this approach is useful. In fact, the integration between R and Hadoop allows to develop the whole of the algorithm in an R environment, by writing in MapReduce only the parts that need scalability.

Regarding the package, there are some possibilities, such as the connectors

provided by Oracle, i.e. the "ORCH" package, and by Revolution Analytics [10], i.e. the "rmr2" package. In the case study, the chosen is rmr2. In addition, the Comprehensive R Archive Network (CRAN) provides "RHive" package that allows the integration with Hive.

Through all these packages, it is possible to develop the most of the analysis procedures in the same environment, obtaining complex ad accurate techniques, e.g. machine learning algorithms. In fact, if the tools are not integrated, the analysis requires the separate and sequential use of them, precluding some possibilities.

2.4.1 "rmr2" package

As mentioned before, the most important R-Hadoop integration package is rmr2. In fact, it deals with the algorithm that is the part of analysis that requires the integration.

The rmr2 package provides "mapreduce" function that allows to write MapReduce jobs, taking the following inputs.

Map function

Reduce function

Combine function

HDFS path of input file

HDFS path of output file

Map, Reduce, and Combine functions take as an input a key-value matrix that is an HDFS chunk. Since the output will be stored in HDFS too, it should be in the key-value form. In detail, all functions are written in R code that will be translated in Hadoop Java code in an automatic way.

An advantage is that all MapReduce function can access to all R variables. This is very useful since many algorithms need the use a small set of information during their steps. For instance, K-means algorithm, described in Chapter 3, requires to keep a list of the centers.

Chapter 3

K-means and variations

The starting point for all centroid-based clustering algorithms is K-means [11]. In fact, it is simple, basic, and easily scalable. This chapter presents some variations of the algorithm that consist in different initialization choices.

3.1 Basic K-means

The subject of the analysis is a set of I objects $x_1, ...x_I$ described by F features each. It means that

$$x_i = (x_i^1, ..., x_i^F), \forall i = 1, ..., I$$

K-means partitions the set $X = \{x_i, i = 1, ..., I\} \subset \mathbb{R}^F$ in the clusters $C_1, ..., C_K$, identified by the centers $c_1, ..., c_K$ respectively. Each center represents the "average object" of the corresponding cluster, i.e. a fictitious object which features are the average of the ones of all the objects of the cluster.

An index that expresses the dispersion within the objects of a cluster is the within-cluster sum of squares, i.e. the sum of the square distances between the objects that belong to the cluster and its center.

The aim is to choose the partitioning that minimizes the sum of the within-cluster sums of squares of all clusters. Let $P = (C_1, ..., C_k)$ be the partition of X. Then, the choice of clusters has the goal of

$$P = \arg\min_P \sum_{k=1,...,K} \sum_{i \in C_k} \|x_i - c_k\|^2$$

where $\|\cdot\|$ is the Euclidean norm

$$\|x\| = \left(\sum_{f=1,\dots,F} \left(x^f\right)^2 \right)^{1/2}$$

K-means is an iterative algorithm since it consists in a repeated update of the centers. First, before starting the process, it is necessary to choose the seeds, i.e. the initial values of the centers [1]. Then, the K-means iterated step consists in first updating the choice of the clusters by assigning each object to the nearest center, then recomputing the center of each cluster. The detailed steps are the following.

1. **Assignment step** Assign any object x_i to the cluster C_k if the center c_k is the nearest.
$$k = \arg \min_{k=1,\dots,K} \|x_i - c_k\|^2, \forall i = 1, \dots, I$$

2. **Update of the centers** For any cluster, compute the value of its center as the barycenter.
$$c_k = \frac{1}{|C_k|} \sum_{i \in C_k} x_i, \forall k = 1, ..., K$$

In order to determine when to stop updating, the best choice is to repeat this step until the algorithm reaches the convergence, i.e. until there is no change in the centers. If the convergence does not happen in a reasonable time, it is possible to stop updating. In order to do that, the most common option is to fix the maximum number of steps before starting K-means, although unfortunately this way does not ensure the convergence.

3.1.1 K-means performances

K-means is the main centroid-based clustering algorithm, but it is not always the best choice.

First of all, K-means is not a stable algorithms, so different seeds choices may lead to different results. Because of this fact, it is essential to choose a proper initialization method. Conversely, in the absence of a good criterion, we can run the algorithm multiple times, using different initializations. In fact, a good understanding of the problem can come from the comparison of different

[1] As regards the initialization, there are different options that will be described afterwards.

results. Since we deal with initialization in Section 3.2, this section assumes that the algorithm has already been properly initialized.

Another limit of K-means is that its cluster assignment is solely based on the distance between the objects and the centers of the clusters. As most of the objects are close to the center, the shape of the groups is spherical-like. In this way, the algorithm cannot identify the weird-shaped cluster. Example 3.1 illustrates this fact, showing a case in which K-means does not succeed in the identification of two clearly distinct clusters.

***Example* 3.1** (Semicircle dataset)**.** *We have two sets of bivariate objects.*

Cluster 1 *The objects are distributed around the center* $(0, 0)$ *and the maximum distance is* $\rho_{max} = 1$*.*

$$\begin{cases} y_i = (\rho_i \sin\theta_i, \rho_i \cos\theta_i) & \\ \rho_i \sim U(0, \rho_{max}) & \forall i = 1, ..., N_2 \\ \theta_i \sim U(0, 2\pi) & \end{cases} \tag{3.1}$$

Cluster 2 *The objects are distributed like a normal which mean is distributed like a uniform around an half-circumference.*

$$\begin{cases} y_i = (\rho_i \sin\theta_i, \rho_i \cos\theta_i) & \\ \rho_i \sim N(\rho_0, \sigma) & \forall i = 1, ..., N_2 \\ \theta_i \sim U(-\frac{\pi}{2}, \frac{\pi}{2}) & \end{cases} \tag{3.2}$$

The parameters are $\sigma = 0.2$ *and* $\rho_0 = 5$

The R implementation is shown in Appendix B.1.1.

After a proper initialization, K-means divides the dataset into 2 clusters. As can be seen in Figure 3.1 (a), the algorithm is not able to identify the centers.

By the way, it is possible to work around the problem. In fact, if K-means is told to divide the dataset in 3 clusters, it finds precisely cluster 1, but splits the curved clusters in two parts (see Figure 3.1 (b)). This fact shows that K-means can recognize the regions, but it needs to split into two parts the weird-shaped cluster.

Example 3.1 shows that K-means is not bad at all. Unfortunately, in some situations, the shape of the clusters can be a considerable problem. Examples 3.2 - 3.3 show the failure of K-means clustering on weird-shaped datasets.

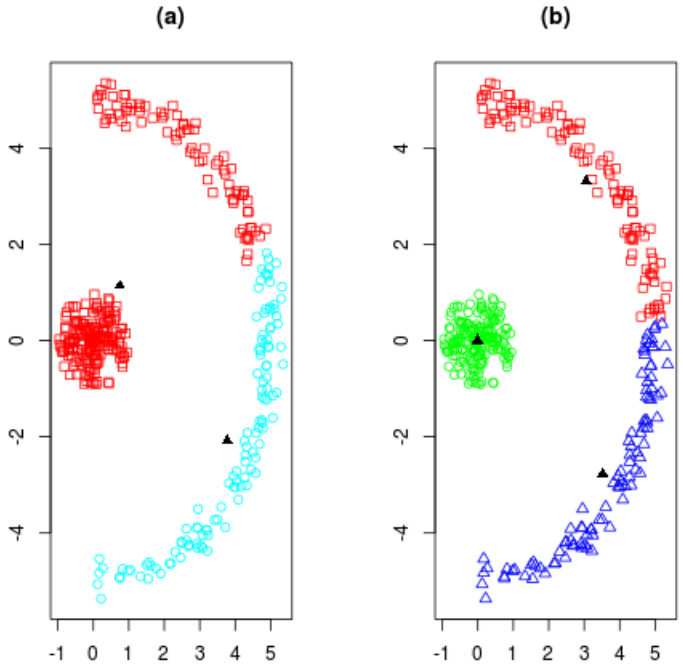

Figure 3.1: Semicircle dataset test on K-means.

***Example* 3.2** (Dartboard dataset)**.** *We have three sets of N bivariate objects.*

Central cluster *The objects are normal-distributed around the center.*

$$\begin{cases} y_i = (\rho_i \sin\theta_i, \rho_i \cos\theta_i) & \\ \rho_i \sim N(\rho_{01}, \sigma) & \forall i = 1, ..., N \\ \theta_i \sim U(-\pi, \pi) & \end{cases} \tag{3.3}$$

Central ring *The objects are distributed like a normal with mean distributed like an uniform around a circumference with radius ρ_{02}.*

$$\begin{cases} y_i = (\rho_i \sin\theta_i, \rho_i \cos\theta_i) & \\ \rho_i \sim N(\rho_{02}, \sigma) & \forall i = 1, ..., N \\ \theta_i \sim U(-\pi, \pi) & \end{cases} \tag{3.4}$$

Peripheral ring *The objects are distributed like a normal with mean distributed like an uniform around a circumference with radius $\rho_{03} > \rho_{02}$.*

$$\begin{cases} y_i = (\rho_i \sin\theta_i, \rho_i \cos\theta_i) & \\ \rho_i \sim N(\rho_{03}, \sigma) & \forall i = 1, ..., N \\ \theta_i \sim U(-\pi, \pi) & \end{cases} \tag{3.5}$$

The parameters are $\rho_{01} = 0$, $\rho_{02} = 2$, $\rho_{03} = 4$, and $\sigma = 0.1$. The R implementation is shown in Appendix B.1.2.

If the fixed number of centers is set to three, K-means does not succeed in the identification of the clusters. Furthermore, even if the algorithm is told to find 10 clusters, it still does not give acceptable results since the identified clusters contain points belonging to different sets. Figure 3.2 shows both results.

***Example* 3.3** (Spiral dataset)**.** *We have two sets of N bivariate objects. The ones of any set are distributed like a normal with mean distributed like an uniform on a spiral.*

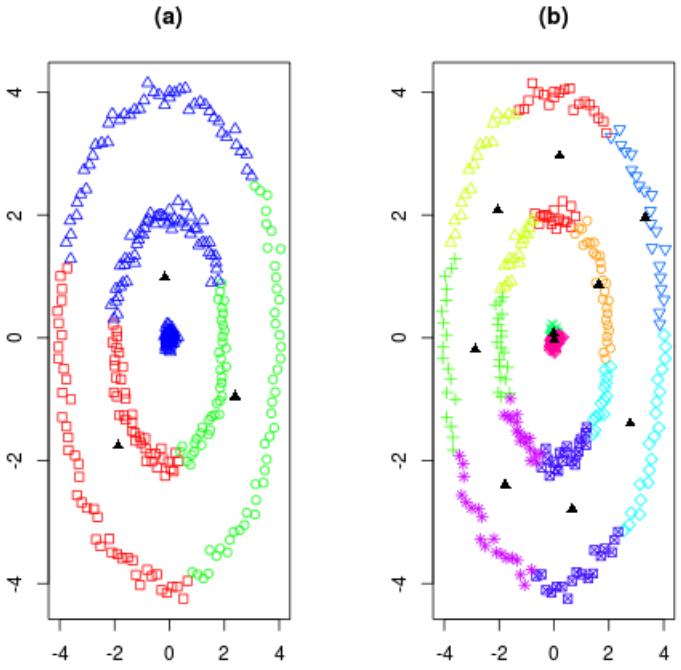

Figure 3.2: Dartboard dataset test on K-means.

Spiral cluster 1

$$\begin{cases} y_i = (\rho_i \sin\theta_i, \rho_i \cos\theta_i) & \\ \rho_i \sim N(5s_i, \sigma) & \forall i = 1, ..., N \\ \theta_i = 4\pi s_i & \\ s_i \sim U(0,1) & \end{cases} \tag{3.6}$$

Spiral cluster 2

$$\begin{cases} y_i = (\rho_i \sin\theta_i, \rho_i \cos\theta_i) & \\ \rho_i \sim N(5s_i + 1, \sigma) & \forall i = 1, ..., N \\ \theta_i = 4\pi s_i & \\ s_i \sim U(0,1) & \end{cases} \tag{3.7}$$

The R implementation is shown in Appendix B.1.4.

As Figure 3.3 shows, the results are even worst than in the case of the Dartboard dataset, since each cluster contains some points that belong to both sets.

Another problem is that, even in situations in which the real clusters are round-shaped, sometimes K-means does not work well. In fact, it has a tendency to generate equally-sized clusters, as shown in Example 3.4

***Example* 3.4** (Mouse dataset)**.** *We have three sets of bivariate data. One of them is big-sized and the other two are smaller.*

Big cluster

$$\begin{cases} y_i = (\rho_i \sin\theta_i, \rho_i \cos\theta_i) + x_1 & \\ \rho_i \sim U(0, \rho_{01}) & \forall i = 1, ..., N_2 \\ \theta_i \sim U(0, 2\pi) & \end{cases} \tag{3.8}$$

Small cluster 1

$$\begin{cases} y_i = (\rho_i \sin\theta_i, \rho_i \cos\theta_i) + x_2 & \\ \rho_i \sim U(0, \rho_{02}) & \forall i = 1, ..., N_2 \\ \theta_i \sim U(0, 2\pi) & \end{cases} \tag{3.9}$$

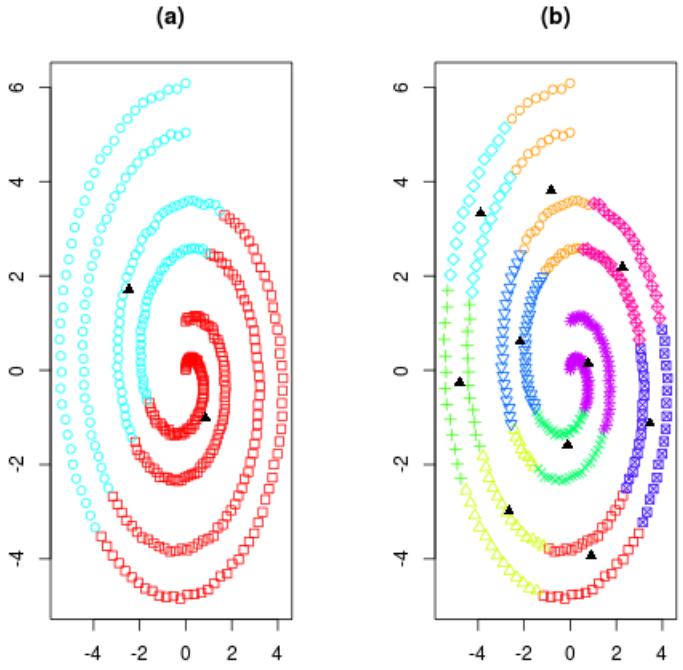

Figure 3.3: Dartboard dataset test on K-means.

Small cluster 2

$$\begin{cases} y_i = (\rho_i \sin\theta_i, \rho_i \cos\theta_i) + x_3 \\ \rho_i \sim U(0, \rho_{03}) \qquad\qquad \forall i = 1, ..., N_2 \\ \theta_i \sim U(0, 2\pi) \end{cases} \tag{3.10}$$

The parameters are $\rho_{01} = 4.5$, $\rho_{02} = \rho_{03} = 2$, $x_1 = (0, 0)$, $x_2 = (5, 5)$, *and* $x_3 = (-5, 5)$

The R implementation is shown in Appendix B.1.3.

As can be seen in Figure 3.4, the algorithm identifies the three clusters, but it assigns some points of the bigger to the two smaller.

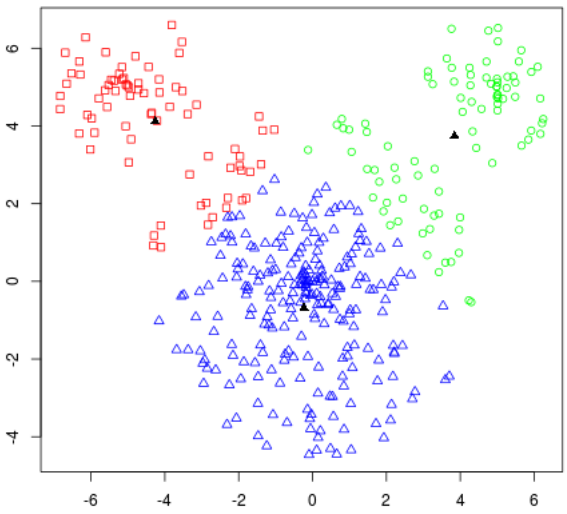

Figure 3.4: Mouse dataset test on K-means.

3.2 K-means initialization

In this section, we will focus on the choice of K-means initial centers, i.e. the seeds. This step is very important, since different initializations may lead to

different results, and there are some alternatives. This section describes some of them.

3.2.1 K-means random partitioning initialization

A possibility is to give the same relevance to all data. It means that the choice is totally random, so it is a naive approach. The only assumption is that the number of clusters K has already been fixed. There are two possible approaches that are random segmentation and random sampling.

Random segmentation splits the dataset in K subsets, by labeling each datum x in a random way. Let $k_x \in \{1, \dots K\}$ be the tag of x that associates it to the cluster k. The computation of the seeds is an average of all objects with the same label k. The steps of the random segmentation are the following.

1. **Label assignment** $k_x = sample(1, ..., K), \forall x \in X$
2. **Clusters definition** $C_k = \{x \in X : k_x = k\}, \forall k = 1, ..., K$
3. **Centers computation** $c_k = \sum_{x \in C_k} x/|C_k|, \forall k = 1, ..., K$

The peculiarity of this initialization is the closeness between the centers. In fact, especially if the dataset is rather wide, all of them will be likely located near the barycenter. This fact conveys in a stability of the seeds since they likely belong to the same region. Unfortunately, in some cases, there is an incongruity between the initialization and the real nature of data, as shown in Example 3.5.

***Example* 3.5** (Four regions dataset)**.** *We have four sets of N normal-distributed bivariate objects.*

Cluster 1

$$x_i \sim N\left(\begin{pmatrix} 1 \\ 1 \end{pmatrix}, \begin{pmatrix} \sigma & 0 \\ 0 & \sigma \end{pmatrix}\right), i = 1, ..., N$$

Cluster 2

$$x_i \sim N\left(\begin{pmatrix} -1 \\ 1 \end{pmatrix}, \begin{pmatrix} \sigma & 0 \\ 0 & \sigma \end{pmatrix}\right), i = 1, ..., N$$

Cluster 3

$$x_i \sim N\left(\begin{pmatrix} 1 \\ -1 \end{pmatrix}, \begin{pmatrix} \sigma & 0 \\ 0 & \sigma \end{pmatrix}\right), i = 1, ..., N$$

Cluster 4

$$x_i \sim N\left(\begin{pmatrix} -1 \\ -1 \end{pmatrix}, \begin{pmatrix} \sigma & 0 \\ 0 & \sigma \end{pmatrix}\right), i = 1, ..., N$$

The initialization algorithm splits the points in 4 clusters, by assigning any of them to a random one. In this way, all clusters contain much the same number of points of each set, so the seeds will be in the central area, where the data are absent, as shown in Figure 3.5. The problem is that this choice is much far from the real nature of the problem and this fact can lead to bad results, even after K-means reaches the convergence. Figure 3.6 shows the final results that come from a multiple run of the algorithm.

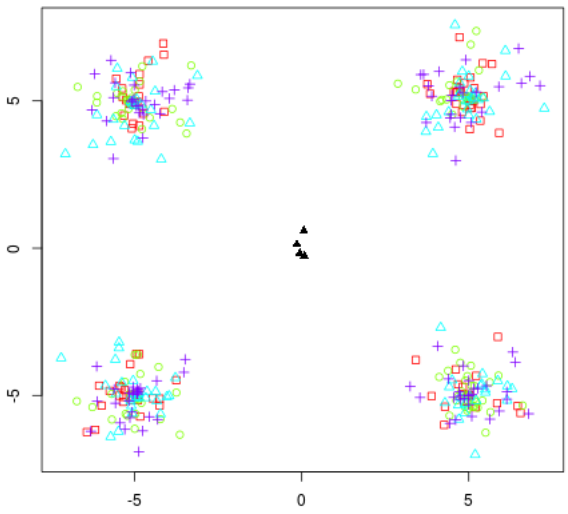

Figure 3.5: Random partitioning initialization, with centers that do not belong to any cluster.

As regards the choice of the number of clusters, random initialization lacks of a proper criterion, so it is often necessary to fix it beforehand. A possibility is to run the algorithm more than once and to compare the results, in order to obtain the correct number.

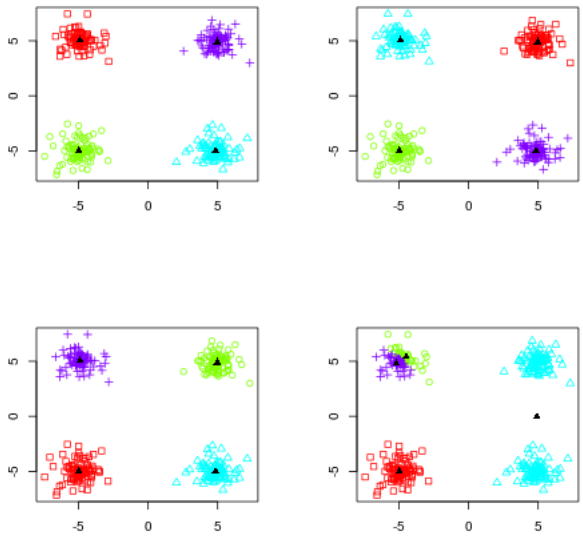

Figure 3.6: Results of K-means run with 4 random partitioning initialization.

3.2.2 K-means random sample

A second chance for a random initialization consists in sampling K random points. This approach [13], i.e. the *random sampling*, is a good alternative to the random segmentation since it makes possible the identification of clusters that are distant from the central region.

The disadvantage is that, because of the multitude of possibilities, the method is much unstable, so it performs badly if it is used just once. Furthermore, it samples each point with the same probability, so it has the tendency of ignoring the small clusters since they contain only a few points. On the other hand, for the same reason, it likely samples more than a single point from the big clusters. Furthermore, even if the size of the clusters is about the same, the randomness of sampling implies an instability of this method. For example, let us suppose there are K clusters of the same size and that the number of seeds is set to K. Random sampling likely chooses more than a single point from a single cluster and no one from others.

However, random sampling initialization can ve very useful in some situa-

tions. In fact, some results can be obtained through the repetition of random-sampling-initialized K-means different times, in order to identify each cluster at least once. It is also useful to vary the fixed number of desired seeds, in order to see how the segmentation changes. In this way, the analysts can explore the nature of data, in order to choose a proper initialization.

3.2.3 K-means++ sampling initialization

In situations in which some clusters are much bigger than the others, an ideal initialization chooses a single seed from any cluster, independently from its size. On the contrary, the random sampling has the tendency of choosing the most of the seeds from the big clusters and random partitioning is inclined to ignore the peripheral clusters.

A possible solution is to sample initial points that are as much as possible scattered among the domain. It means that, if two points are near, the algorithm will unlikely sample both of them. In this way, it is unlikely a situation in which two points will belong to the same cluster, unless it is very big. On the other hand, if there is a small cluster that is distant from the others, the algorithm will likely sample a point from it.

An initialization that is based upon this logic is K-means++ [14]. It consists in a one-by-one sampling of seeds, in such a way that each new seed is likely as much as possible distant from all the previous ones. The idea is that each point can be potentially sampled, but the sampling probability is proportional to the squared distance between the nearest seed and it.

The algorithm consists in the iteration of the same step, until the number of seeds reaches K. Let X be the set of all points, C the set of already chosen seeds, and $D : X \longrightarrow \mathbb{R}$ a function that defines the distance between a generic object and its closest seed.

$$D(x) = \min_{c \in C} \|x - c\|, \forall x \in X$$

First, the initial point is sampled in a totally random way, i.e. $c_1 = sample(X)$ with uniform probability.

Then, the other points are sampled until the number of the seeds reaches K. It means that, for any $k = 2, ..., K$, the algorithm iterates the following steps.

1. Computation of probabilities $p_x = \frac{D^2(x)}{\sum_{y \in X} D^2(y)}, \forall x \in X$

2. Sample $c_k = sample(X)$ with probability $p = \{p_x, x \in X\}$

Figure 3.7 shows an example of K-means++ initialization on a testing dataset, in which there are four distinct different-shaped clusters.

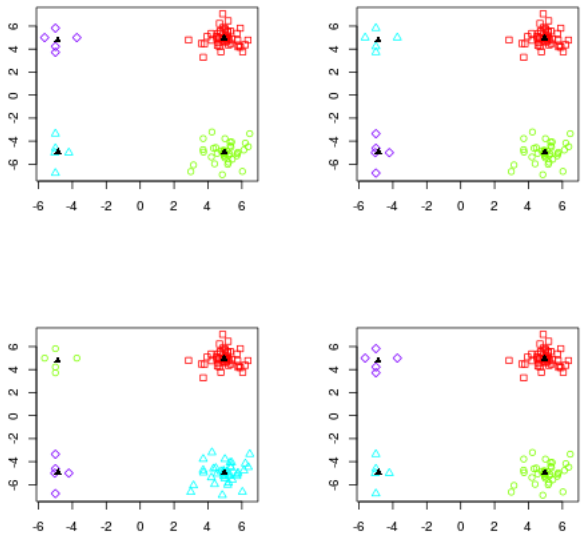

Figure 3.7: Results of K-means++ initialization repeated 4 times.

3.2.4 K-means Kauffman initialization

Despite K-means++ initialization improves the K-means performances, it does not always choose the same seeds. In fact, randomness comes from the fact that each point can potentially be sampled, although with different probabilities.

In order avoid this fact, it is possible to initialize K-means in a deterministic way through the Kaufman initialization [12]. In fact, it samples the seeds one at a time, by choosing the point that is as much as possible distant from the already sampled ones.

Since the algorithm will use all distances between two objects, the value of all of them are computed before.

$$d(x, y) = \|x - y\|, \forall x, y \in X$$

The steps of the algorithm are the following.

1. **First sampling** Selection of the point c as the most centrally located, i.e. the closer to the mean of all points, and addition of it to the list of the centers.

$$c = \arg\min_{x \in X} \left\| x - \frac{1}{|X|} \sum_{x \in X} x \right\|$$
$$C = \{c\}$$

2. **Iterative sampling** While the number of seeds is less than K

   2a. Computation of the minimum distances

$$D(x) = \min_{c \in C} d(x, c), \forall x \in X \setminus C$$

   2b. Computation of the indexes

$$b(x, y) = \max(D(x) - d(x, y), 0), \forall x, y \in X \setminus C$$

   2c. Choice of the new seed

$$c = \arg\min_{x \in X} \sum_{y \in X} b(x, y)$$
$$C = C \cup \{c\}$$

Figure 3.8 shows that Kaufman does not succeed in sampling a point from each one of the 4 clusters. As a matter of fact, the smallest is ignored and two points are sampled by the biggest. However, if the fixed number of seeds is set to 5, results are better, as shown in Figure 3.9. In fact, it samples at least a point from any cluster.

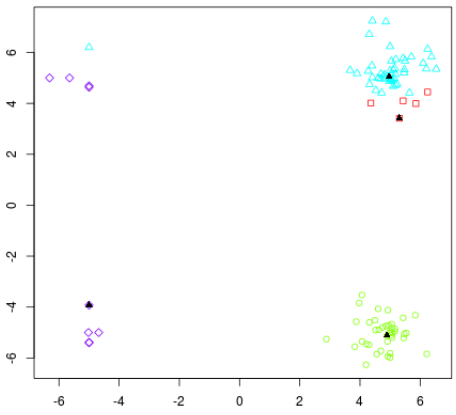

Figure 3.8: Kaufman sampling of 4 points.

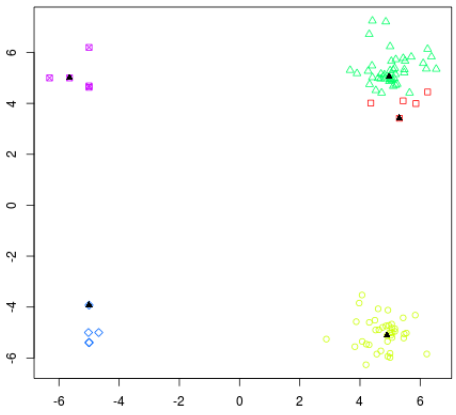

Figure 3.9: Kaufman sampling of 5 points.

Chapter 4

Hybrid clustering

As regards clustering, it can be based on connectivity or on centroids. The fist case is called *hierarchical* and the other is *centroid-based*.

Hierarchical clustering is very slow and the computational cost explodes as the data volume grows. Furthermore, the algorithms need to build and use huge matrices, so the volume easily exceeds the capacity of devices, even in a Big Data framework. For these reasons, hierarchical clustering algorithms cannot resolve the most of Big Data problems themselves.

On the other hand, centroid-based clustering provides some fast and scalable algorithms that do not need the use of big matrices. For this reason, they are a good starting point for a Big Data analysis, but they often lack of efficiency. The challenge is to use a new hybrid method that combines the two approaches, maintaining both scalability and effectiveness as well as possible.

Although Hierarchical clustering techniques are not a good choice as regards many Big Data analysis, they play an important role in the development of hybrid approaches. For this purpose, Section 4.1 gives a brief overview of some techniques that take part in the hybrid algorithm. The other sections, instead, describe two possible hybrid clustering techniques.

4.1 Hierarchical clustering

Hierarchical clustering is based upon the logic of first taking account of many partitioning options, then choosing the most proper one. As regards the possible partitions, the two extremities that are a single cluster containing all the objects

and a cluster for each object. In order to define the set of options, there are two main approaches.

Agglomerative clustering First, each object belongs to its own cluster. Then, the clusters are merged, according to the similarity between them.

Divisive clustering First, all the objects belong to the same cluster. Then, the algorithm iteratively splits the clusters.

The approach that takes part in the development of a hybrid algorithm is agglomerative clustering, so this section is only about it.

First of all, let us give a distance $d : X \times X \to \mathbb{R}$ between two objects, e.g. the Euclidean norm. In order to define an evaluation criterion of similarity between two clusters, it is necessary to define also the distance $D : \mathcal{P}(X) \times \mathcal{P}(X) \to \mathbb{R}$ between two subsets. For this purpose, there are three main options.

Single linkage Given two clusters C_1 and C_2, the distance is the minimal distance between an object of C_1 and an object of C_2.

$$D(C_1, C_2) = \min_{x_1 \in C_1, x_2 \in C_2} d(x_1, x_2)$$

Complete linkage It is the opposite of single linkage since it uses the maximal distance.

$$D(C_1, C_2) = \max_{x_1 \in C_1, x_2 \in C_2} d(x_1, x_2)$$

Average linkage It is a halfway between the two other two options and the distance is the mean of all the distances.

$$D(C_1, C_2) = \frac{1}{|C_1||C_2|} \sum_{x_1 \in C_1, x_2 \in C_2} d(x_1, x_2)$$

Once the linkage has been chosen, the algorithm iteratively merges the two nearest clusters. For this purpose, the distances between all couples of objects are put in a matrix, which rows and columns correspond to the objects. This distance matrix is symmetric and the main diagonal contains “0" values. Starting from that, it is easy to compute the distance between each pair of clusters, according to the linkage, and to identify the minimum.

After some iterations of this process, all the objects will belong to the same cluster, so it is necessary to stop before it happens. For this purpose, there are two possible options.

Number of clusters The process goes on, until the number of clusters is less than or equal to a fixed one.

Distance threshold The process goes on as long as there is at least a pair of clusters which distance is less than a fixed threshold.

In order to determine a good criterion, it is possible to build a tree diagram, called dendogram, that shows the progression of the agglomeration. At the bottom of the diagram, each object belongs to a different cluster. Then, going up, in each level the merge affects all the clusters which distance is below the level. The number of clusters or the distance threshold can be deduced by the observation of the dendogram.

Figure 4.1 shows an example, in which it is easy to deduce that the number of clusters is equal to 2 or 4. In fact, starting from the top, the dataset is initially split into 2 clusters and immediately after in 4. Afterwards, the splitting is more confusing. As regards the distance threshold, it is 0.8, if the clusters are two, or 0.6, if the clusters are four.

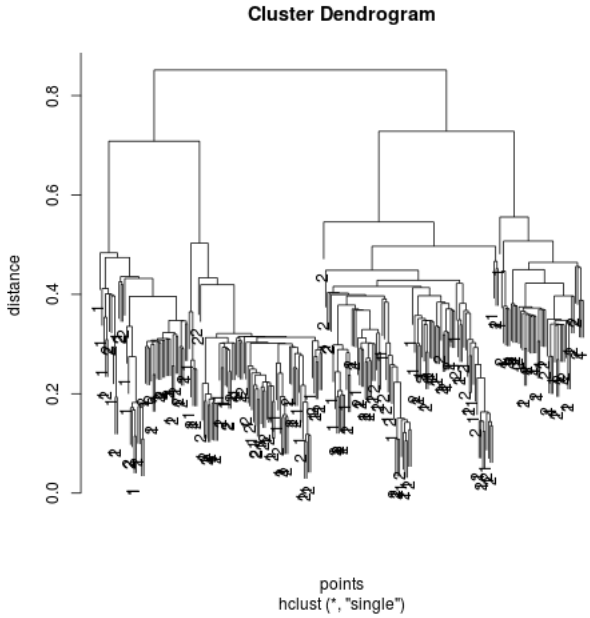

Figure 4.1: Example of a dendogram.

The results of the clustering depend on the choice of both the distance function and the linkage method (single, complete, or average). As regards the distance function, if data are numerical, the most common option is the Euclidean distance. The linkage choice, instead, is more complicated, as it depends on how data are distributed. In order to show the differences in their effectiveness, hierarchical clustering with different linkages has been tested on the following four datasets, described in detail in Section 3.1.

Mouse dataset The objects belong to a big round cluster and to two smaller ones.

Semicircle dataset There are a round cluster and a bowed one.

Dartboard dataset There are a central round cluster and two concentric circular ones around it.

Spiral dataset There are two spiral clusters.

Single linkage is the best choice in the absence of noise and it is a useful tool in the search for weird-shaped clusters. As shown in Figure 4.6, it succeeds in identifying all the clusters. Conversely, complete linkage has some problems in the identification of the weird-shaped ones. Figure 4.11 shows that the complete linkage gives the correct results about the Mouse dataset only.

The algorithms have been tested also on the same datasets with the addition of some noise, i.e. of points that are randomly scattered around the clusters. Unfortunately, single linkage does not work well in the presence of some doubt points that do not belong to any clearly distinct cluster, as shown in Figure 4.16. In fact, the algorithm merges two clusters even if there are only a few points that make a bridge between them. Fortunately, complete linkage is more stable, as it provides slightly better results, although not yet satisfactory. Figure 4.21 shows that the algorithm succeeds in eliminating the outliers groups, but it is still not able to identify the right clusters.

As regards average linkage, it has been tested, but it is affected by both problems, so the results are not reported.

In conclusion, hierarchical clustering has a lot of problems in the identification of clusters in the presence of outliers. In addition, the methods are not scalable. A possible solution to these problems is a hybrid approach.

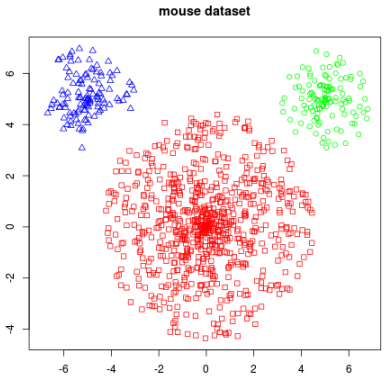

Figure 4.2: Mouse dataset.

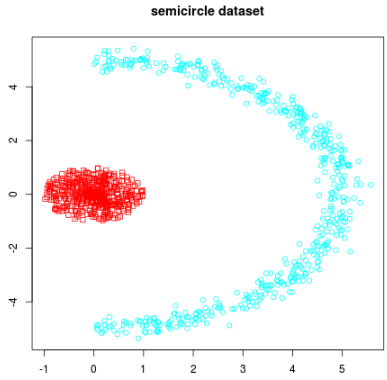

Figure 4.3: Semicircle dataset.

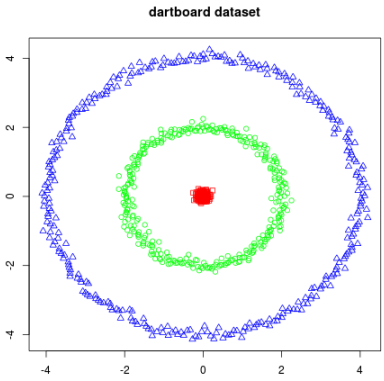

Figure 4.4: Dartboard dataset.

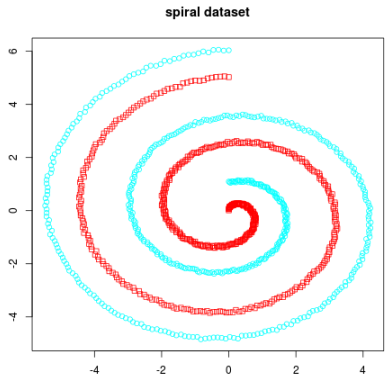

Figure 4.5: Spiral dataset.

Figure 4.6: Single-linkage hierarchical clustering testing.

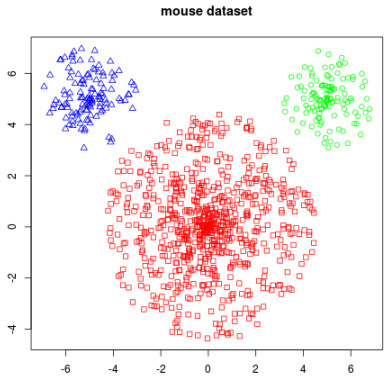

Figure 4.7: Mouse dataset.

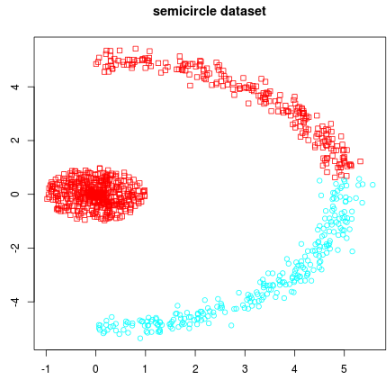

Figure 4.8: Semicircle dataset.

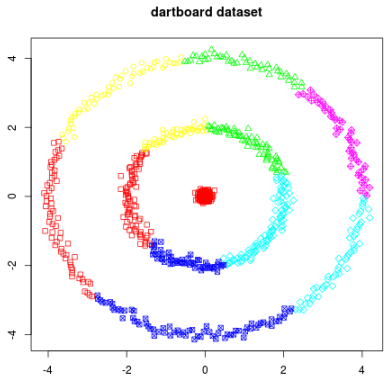

Figure 4.9: Dartboard dataset.

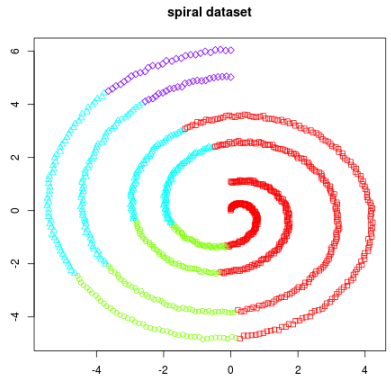

Figure 4.10: Spiral dataset.

Figure 4.11: Complete-linkage hierarchical clustering testing.

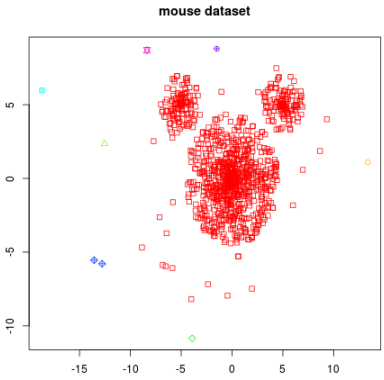

Figure 4.12: Mouse dataset.

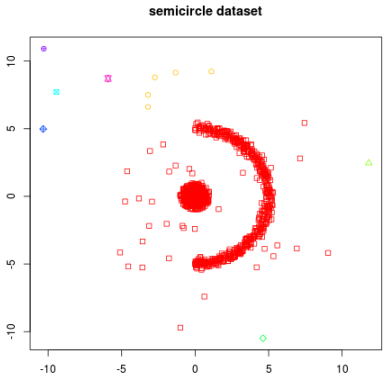

Figure 4.13: Semicircle dataset.

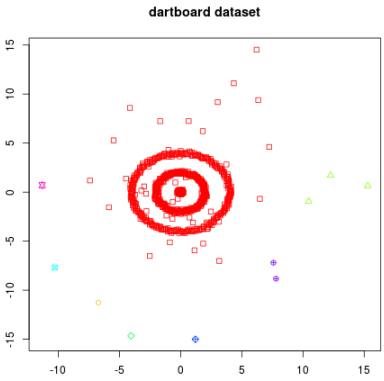

Figure 4.14: Dartboard dataset.

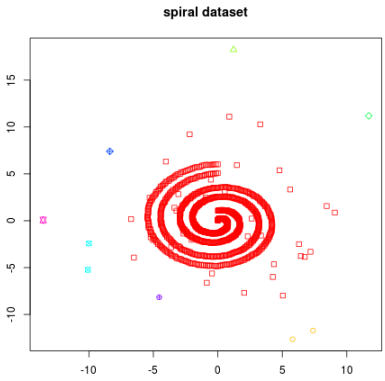

Figure 4.15: Spiral dataset.

Figure 4.16: Single-linkage hierarchical clustering testing on datasets with noise.

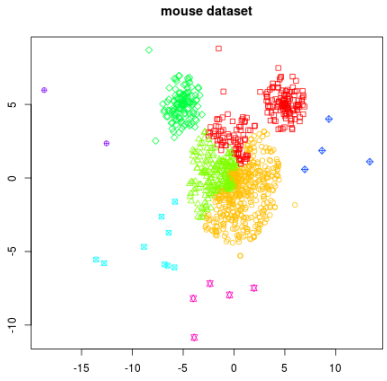

Figure 4.17: Mouse dataset.

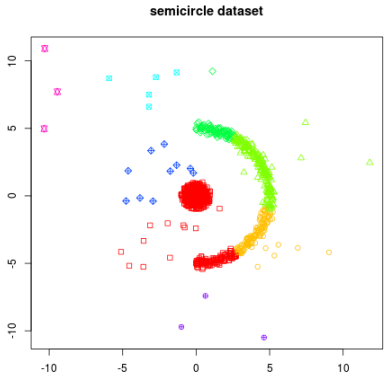

Figure 4.18: Semicircle dataset.

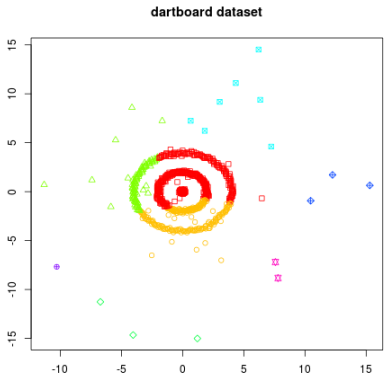

Figure 4.19: Dartboard dataset.

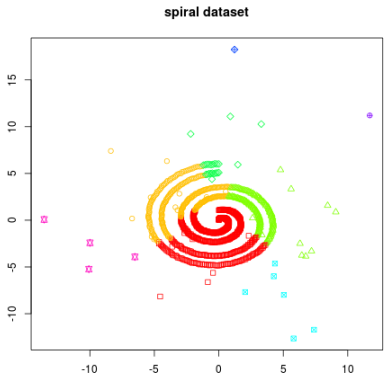

Figure 4.20: Spiral dataset.

Figure 4.21: Complete-linkage hierarchical clustering testing on datasets with noise.

4.2 Split-merge algorithm

The Split-merge algorithm combines the efficiency of centroid-based clustering techniques with the effectiveness of hierarchical approaches. In order to do that, it first reduces the size of the problem with a scalable algorithm, then applies hierarchical clustering on the reduced dataset. The overall method consists in the following procedures.

Dataset split K-means-like clustering, obtaining a wide number of clusters.

Hierarchical merge Merge of the centers of the clusters, using a hierarchical algorithm.

Clusters identification Assignment of the objects to the clusters defined by the merge.

The procedures are illustrated in the next subsections.

4.2.1 Dataset split

The purpose of this procedure is to obtain small and very homogeneous clusters. In this way, each point is represented by the nearest cluster center in a precise way.

A possibility is to identify a wide number of clusters using through K-means. Nevertheless, this approach leads to the following problems.

Initialization In order to obtain the stability, K-means should be initialized in a proper way. As mentioned in Chapter 3.2.3, a stable method is K-means++. However, its computational cost explodes if the desired number of centers is high, so it is not a good choice in this case.[1]

Outliers sensitivity K-means is much sensitive to outliers, especially if any cluster contains only a few points. As a matter of fact, the center of any small cluster is dragged by a single outlier.

Outlier groups detection If there are some useless outliers located in the same region, K-means will likely identify a cluster that contains them. The problem is that they can cause some troubles during the hierarchical merge, as shown in Subsection 4.2.2.

[1] This problem is partly solved by a scalable version of K-means++ initialization, shown in Subsection 5.2.2

Useless split If there is a compact cluster with many points, it will be separated into many sub-clusters, as K-means always tries to find out the fixed number of clusters. Although the hierarchical merge solves this problem, it is very inefficient if the number of sub-clusters is too high.

A solution to those problems consists in dividing the split procedure in two parts.

1. Split the objects in a small number of clusters.

2. Iteratively split any cluster and detect the outliers.

First, it is necessary to define a list containing the centers of all the already chosen clusters. This list is empty at the beginning and is iteratively updated.

As regards the first part, it can be done using K-means, initialized with K-means++. Now, the list of the centers is composed by in the ones of the identified clusters. In order to avoid having centers that are too near, the ones that are too close are removed. A fast in-memory algorithm computes the distance between any couple of centers and, if it is below a fixed threshold, removes a center from them.

The second part consists in iterating the following steps for a fixed number of times.

1. **Addition of the centers** For any cluster, the object that is the most distant from the center is added to the list, under the condition that the distance between the center and it is more than a fixed threshold. This condition avoids the split of small-sized clusters.

2. **Update of the centers** In order to fix the centers, the algorithm performs a single K-means step and updates the list of the centers.

3. **Removal of the outliers** This step consists in the count of the objects belonging to any cluster. If the size of a cluster is below a fixed threshold, the corresponding center is removed from the list.

4. **Assignment to the centers** Any object that belonged to a removed cluster is assigned to the nearest center.

4.2.2 Hierarchical centers merge and clusters identification

As regards hierarchical clustering, there are three options, i.e. single, complete, and average linkage, which features has been described in Section 4.1. A problem of the techniques is the presence of outliers, but the dataset split has already removed them in order to avoid that.

As regards the linkage, the best choice is usually the single one. In fact, since the outliers has already been removed, the main challenge is to identify the weird-shaped clusters and single-linkage advantage is its effectiveness in that.

This procedure cannot be automated because of the necessity to determine the dendogram cut. Fortunately, the quickness of in-memory analysis allows the trial of some alternatives.

Now, each cluster is identified by the list of its centers. In order to obtain information about them, the algorithm computes some summaries of the features of the clusters. Some possible results are the following.

Size For any cluster, its size is computed by adding together the numbers of the objects belonging to all the centers associated with it.

Mean The mean of any cluster consists in its barycenter and its computation consists in a weighted mean, where the values are the centers and the weights are the number of objects assigned to them.

Variance The variance expresses the dispersion of the objects within any cluster.

4.2.3 Effectiveness evaluation

The methods have been tested on some small-sized datasets using in-memory devices, in order to evaluate their effectiveness. The datasets have already been described in Subsection 3.1.1 and they are the following.

Mouse dataset

Semicircle dataset

Dartboard dataset

Spiral dataset

Furthermore, the datasets contain also some random points, which addition is described by the noise rate, defined as the ratio between the number of random points and the number of points that belong to the clusters. The value of the noise rate is set to 0.05 in all cases.

In all the figures that show the results, the centers that come from the split step are represented by black round-shaped points and the final clusters are identified by different colors and shapes.

The first testing is about the algorithm that uses only K-means during the split procedure. Although this approach leads to some problems, it is useful to compare it with the others. The minimum required cluster size is set to 5 in the case of the Mouse, Dartboard, and Spiral datasets and to 10 in the case of the Semicircle dataset.

As shown in Figures 4.23 - 4.24, the results about the Dartboard and Semicircle datasets are satisfactory. The algorithm performs rather well also in the case of Mouse dataset, with the exception of a small part of the big cluster, as shown in Figure 4.22. However, the algorithm fails in identifying the two spiral-shaped datasets since the most of the centers is concentrated in the central region. Figure 4.25 shows this wrong result.

The testing of the iterative-split variant has been conducted using the same fixed minimum required size. In this case, the results are more satisfactory since the algorithm succeeds in identifying all the clusters. Figures 4.27- 4.30 show the results.

As can be seen, if the parameters have properly been initialized, the algorithm succeeds in identifying the clusters and detecting the outliers. The goodness of results derives from the identification of small sub-clusters that cover the regions of the clusters in their whole.

4.3 Split-merge Same-step variation

The basic Split-merge algorithm divides the split procedure in two parts, due to the necessity of detecting the outliers. The first part only splits the dataset, while the purpose of the second one is to identify the outliers by removing any object that belongs to a small cluster.

An alternative consists in the use of a split step that can both divide the dataset and detect the outliers. At the beginning, all objects belong to the same cluster. Then, the algorithm iteratively splits any cluster in two parts

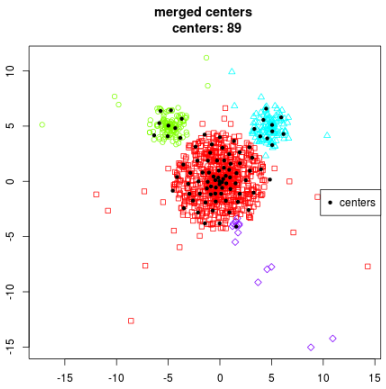

Figure 4.22: Mouse dataset

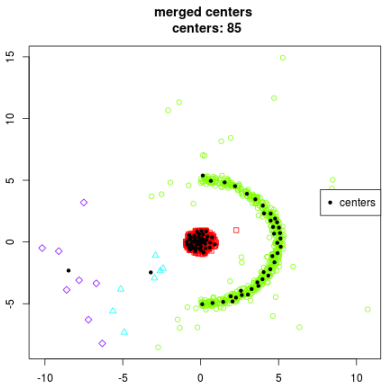

Figure 4.23: Semicircle dataset

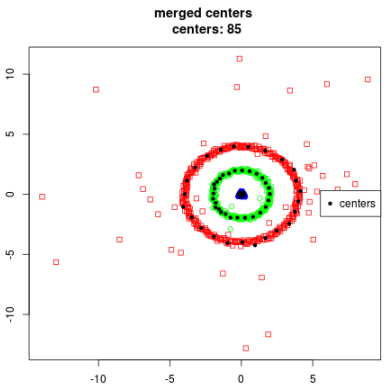

Figure 4.24: Dartboard dataset

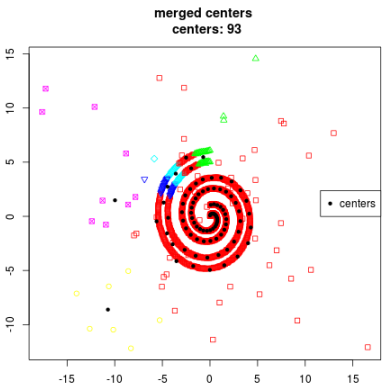

Figure 4.25: Spiral dataset

Figure 4.26: Only-K-means Split-merge clustering testing.

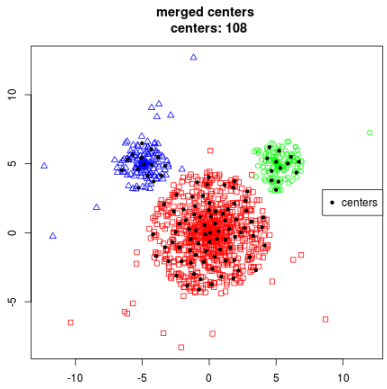

Figure 4.27: Mouse dataset

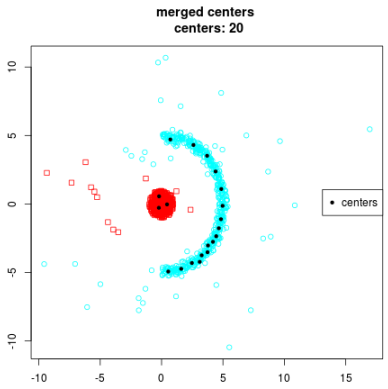

Figure 4.28: Semicircle dataset

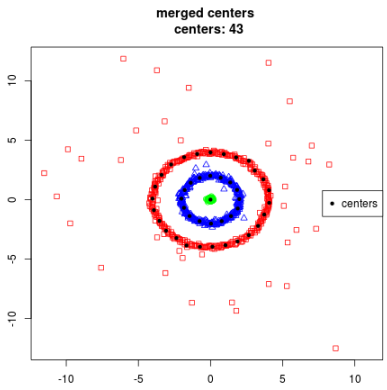

Figure 4.29: Dartboard dataset

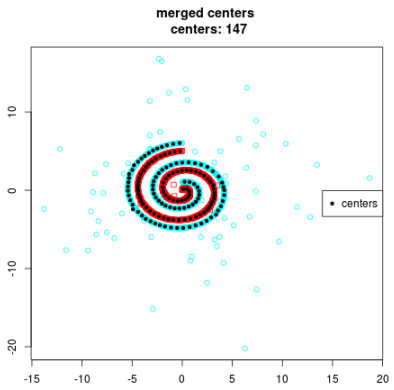

Figure 4.30: Spiral dataset

Figure 4.31: Split-merge clustering testing.

through the same step. The overall procedure, i.e. the combination between this kind of split and the merge, is similar to the one of another hybrid clustering algorithm [16]. The difference is that this new method treats differently small-sized clusters, in order to detect the outliers.

The algorithm treats any cluster that comes from the previous step in a different way, depending on its size. The options are the following.

Option 1. Split into two equally important parts If the cluster is big, K-means splits it in two parts. As regards the initialization, it is possible to use K-means++ without a high computational cost since the number of centers is just 2. By the way, it is also possible to use the other initialization methods without particular problems.

Option 2. Outliers separation If the cluster is medium-sized, the algorithm tries to detect a group of outliers. First, it computes the barycenter and the object that is as much as possible distant from it. If their distance is above a fixed threshold, the cluster is split into two parts, each one of which contains the points that are closer to a center than to the other. After that, a single k-means step fixes the centers.

Option 3. Removal of the outliers If the cluster is too small, its elements are classified as outliers.

Let us fix the parameters, in order to determine how the algorithm will act.

Minimum size of big clusters If a cluster size is above this threshold, the algorithm chooses Option 1.

Maximum size of small clusters If a cluster size is below this threshold, the algorithm chooses Option 3.

Minimum distance that implies the splitting for detecting outliers In Option 2, if the distance is above this threshold, the cluster is split into two parts.

This method performs well when the clusters have different features and problems. In fact, each split step analyzes any cluster singularly, in order to fix the errors that come from the previous step. For example, if a cluster contains two distinct sets of points, a split step separates them.

The main lack of this method is the necessity for a proper parameter choice. In fact, it is not easy to determine whether a cluster is big or small, or when two points are distant enough to separate them.

After the split procedure, the algorithm performs the same merge procedure as the basic version.

4.3.1 Performances

The algorithm has been tested through in-memory devices in order to evaluate its effectiveness. The testing datasets are the same as in Subsection 4.2.3, but the noise rates have been slightly increased, due to an improvement in the performances.

Mouse dataset Noise rate: 0.05. The results are shown in Figure 4.32.

Semicircle dataset Noise rate: 0.1. The results are shown in Figure 4.33.

Dartboard dataset Noise rate: 0.1. The results are shown in Figure 4.34.

Spiral dataset Noise rate: 0.1. The results are shown in Figure 4.35.

Similarly to the other tests, the centers that came from the split step are represented by black and round-shaped points and the final clusters are identified by different colors. As can be seen, the results are satisfactory in all situations.

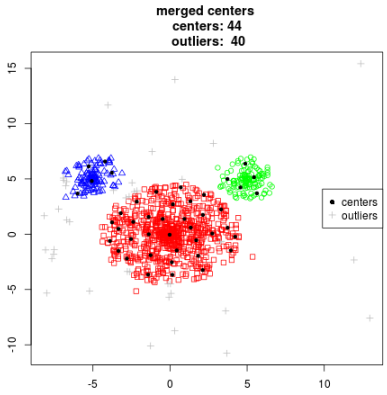

Figure 4.32: Mouse dataset

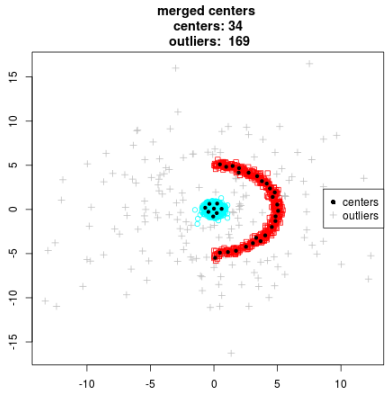

Figure 4.33: Semicircle dataset

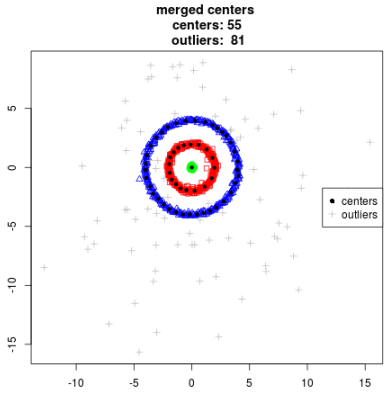

Figure 4.34: Dartboard dataset

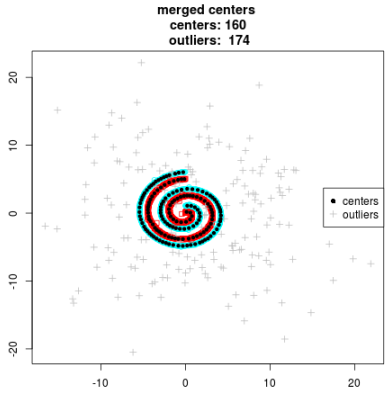

Figure 4.35: Spiral dataset

Figure 4.36: Same-step Split-merge clustering testing.

Chapter 5

Clustering algorithms in MapReduce

In order to perform clustering on Big Data, it is necessary to adapt the clustering methods to the new environment. It means that all the algorithms should be developed in MapReduce, to obtain their scalability on distributed data.

Fortunately, some cluster algorithms are parallelizable, so it is possible to translate them in MapReduce terms. Furthermore, some not scalable algorithm can be slightly modified, in order to have a scalable implementation. Others, instead, cannot be parallelized at all.

This chapter describes the MapReduce implementation of the algorithms. As regards the software, the algorithms have been implemented using R-Hadoop connector rmr2 and the code is reported in Appendix A.2 and in Appendix A.4.

5.1 K-means in MapReduce

The K-means procedure consists in the iteration of the following 2 steps.

Assignment step For any object, compute the distances between the centers and it. Then, assign it to the center that achieves the minimum of all computed distances.

Center update For any cluster, compute the mean of all objects assigned to it.

The assignment step consists in an independent operation for any object, so the process is easily scalable. If any center is associated with a different key, this step is a match of any object with the nearest center key, so it consists in a Map. As regards the update of the centers, any operation is about a single cluster. It means that it affects all the objects having the same key, so it is a Reduce operation.

Furthermore, the Reduce step consists in the computation of a mean that is a scalable operation. The addition of Combine step is not possible because "mean" operation is not associative, unless it is a weighted mean that is both commutative and associative. In order to do that, it is sufficient to associate any object with an extra feature, i.e. a value that represents the weight. During the Map step, the value of the weight of any object is set to "1", as it expresses the fact that it is just one object. Then, during the Combine step, the worker computes a mean and associates any result with a weight equal to the number of objects with the same label within the chunk. It means that the elements of the "1" column are simply summed up. Finally, also Reduce consists in a weighted mean.

Since all the weights are equal to "1", the Combine step consists in an unweighted mean and the "1" column is totally useless. For this reason, a more efficient alternative consists in adding the weight column only after the Combine step, by counting the number of objects.

In detail, the MapReduce algorithm consists in the following steps.

***MapReduce* 1.** *K-means step.*

Map *The worker computes the distances between any object and the centers. Then, it labels the object by associating the key that corresponds to the center which distance is lower. The value consists in the list of the features of the object.*

Combine *For any key, the worker computes the mean of any feature of the objects labeled with it and it computes a weight as the count of objects. The key is the same and the value is the mean associated with the weight.*

Reduce *For any key, the master computes the weighted mean of any feature from the weights. If the Combine step is absent, instead, the master computes an unweighted mean.*

The output consists in the list of the centers.

5.2 K-means initialization in MapReduce

The parallelization of any K-means step is rather easy. However, as mentioned before, the algorithm requires a proper initialization. The challenge of this section is to develop the initialization algorithms in the MapReduce paradigm.

5.2.1 Random choices

As regards random partitioning initialization, it is similar to the K-means step, so it is parallelizable through MapReduce. In this case, the Map step consists in the choice of a random key for any object, in order to assign it to the corresponding cluster. Then, the Combine and Reduce steps are the same as the ones of K-means.

***MapReduce* 2.** *Random-partitioning K-means initialization.*

Map *The worker maps any object with a key that is a random integer between 1 and the fixed number of seeds.*

Combine *For any key, the worker computes the mean of any feature of the objects labeled with it and it computes the weight, i.e. the count of the objects. The value consists in the mean, associated with the weight.*

Reduce *For any key, the master computes a weighted mean. If the algorithm is Combine-less, the mean is unweighted.*

The output consists in the seeds list.

The initialization based on random sampling is rather easy as well. As a matter of fact, it is sufficient to split the objects in random chunks and to sample a single object from each one of them. Splitting takes place during the Map step, while sampling is during the Combine and Reduce steps. The implementation is the following.

***MapReduce* 3.** *Random-sampling K-means initialization.*

Map *The worker maps any object with a key that is a random integer number between 1 and the number of seeds.*

Combine *For any key, the worker samples a random object.*

Reduce *For any key, the master samples a random object.*

The output consists in the seeds list.

Using this MapReduce algorithm, the assignment to the keys is totally random. Hence, it is possible that one or more keys do not have any point associated with them. In this way, the number of seeds is lower than the fixed one, as some keys have not any object associated with them. Fortunately, that situation is much unlikely if the volume of the dataset is wide. Furthermore, random sampling is mostly used by repeating it more than once, in order to obtain different results to compare, so it is not that a problem if the number of seeds is less than the desired one.

5.2.2 K-means++ sampling

Unfortunately, K-means++ is not completely scalable. In fact, sampling is done once at a time and each step depends on the previous ones. It means that any sampling step needs at least an independent MapReduce job. Hence, it is fast only if the desired number of seeds is low.

The initial seed comes from a random sampling that can be performed through MapReduce 3 described before. For any step, the sampling consists in first assigning sampling probabilities, then sampling according to them. As regards the probabilities, their computation needs the sum of all distances. Fortunately, it is sufficient to perform a separate normalization in each step.

The idea is that each object is matched with a weight, proportional to the minimum squared distance between a center and it. Then, it will be sampled by a weight-proportional probability.

***MapReduce* 4.** *K-means ++ single sampling.*

Map *The worker computes the squared distances between any object and the seeds. Then, the value consists in the object features, with the addition of a column containing the minimum squared distance. Since all objects will take part in the same sampling, the key is the same for all.*

Combine *The worker normalizes the weights by dividing each one of them by their sum. Then, it samples an object from these normalized weights. The value is composed of the object features and a new weight, given by the sum of all the weights.*

Reduce *The master divides any weight by the sum of all of them. Then, it samples a single object from the normalized weights. This is the value of*

the new chosen seeds.

The output consists in the sampled object.

Instead of parallelizing the sampling of a single point, it is possible to use a scalable version of the K-means++ [15] that consists in the following two steps.

1. **Sampling of more points than necessary** Sample $K_S \geq K$ points that are as much as possible distant one from each other.

2. **Seeds choice** Cluster the sampled points into K cluster. Then, fix the seeds as the centers of the clusters.

Before describing the algorithm, we specify the notations. Let X be the dataset and let C be the set of already sampled seeds. There are a fixed parameter l that is proportional to the desired number of centers and the following two functions.

The minimum distance between any object and C

$$d : X \to \mathbb{R}$$

$$d(x) = \min_{c \in C} \|x - c\|^2, \forall x \in X$$

The sum of the above minimum square distances

$$\phi_X(C) = \sum_{x \in X} d^2(y, C) = \sum_{x \in X} \min_{c \in C} \|y - c\|^2$$

The algorithm consists in the following steps.

1. **Random sampling of a single point** $C = \{sample(X)\}$

2. **Computation of the cost ψ** $\psi = \phi_x(C)$

3. **Sampling of the other points** For $O(log(\psi))$ times, sample any point independently with probability $p_x = l * d^2(x, C)/\phi_X(C)$

4. **Computation of the weights** For any sampled point $x \in C$, compute the weight ω_x as the sum of points that are closer to x than to any other sampled point.

5. **Clustering of the sample** Cluster the weighted sampled points into K clusters.

6. Seeds choice Compute the seeds as the weighted centers of the clusters.

The first sampling is easy parallelizable through MapReduce, as shown in Subsection 5.2.1, and the computation of the cost ψ requires first to compute a value for any point, then to sum up the values. Hence, it is easily parallelizable too, as shown in MapReduce 5.

***MapReduce* 5.** *K-means++ Ψ computation.*

Map *The worker computes the square distances between any object and the seeds. Then, the value is the smallest squared distance and the key is the same for all the objects.*

Combine *The worker sums up all the chunk values.*

Reduce *The master sums up all the values.*

The output consists in the value of ψ.

As regards sampling, the Map step consists in a computation of probabilities and in a choice of sampling any object or not, while the Reduce step is an ensemble of sampled points. The procedure in shown in MapReduce 6.

***MapReduce* 6.** *K-means++ sampling.*

Map *For any object, the worker computes its sampling probability and it generates a random number between 0 and 1. If the number is less than the probability, the worker uses it as the value of an output. Otherwise, the object is ignored. As regards the key, it is the same for all outputs.*

Combine *The worker generates a matrix with all objects.*

Reduce *The master binds the matrices.*

The output consists in the list of sampled objects.

Next, the seeds must be chosen from the set of sampled points. Fortunately, their number is small, so the algorithm can run in-memory. This algorithm can be K-means, hierarchical clustering, or another one.

5.2.3 Kauffman approach

As regards Kauffman sampling, there is a bottleneck. In fact, if there are n objects, the computation of the indexes $b(x, y)$ implies the storage of $O(n^2)$ values. Clearly the computational cost is too high and the required storage too wide. For example, if there are a million objects, it is necessary to compute a trillion values and to store them.

5.3 Split-merge in MapReduce

In this section, we disclose our MapReduce implementation of the Split-merge algorithm. The following list describes the steps and explains how they are performed. As regards the implementation of any single MapReduce step, it is described afterwards.

First split The initialization consists in the split of the dataset, using K-means. As shown in Sections 5.1-5.2, the algorithm supports the MapReduce implementations, with random partitioning, random sampling, or K-means++ initialization.

Addition of the centers Performed in MapReduce, as shown in MapReduce 7.

Update of the centers Performed in MapReduce, in the same way as a K-means step, as shown in MapReduce 1.

Removal of the outliers Performed in MapReduce, as shown in MapReduce 8.

Assignment to the centers Incorporated in the addition of the centers, as shown in MapReduce 7.

Hierarchical clustering Due to the littleness of the set of centers, it is possible to perform this step in-memory.

Computation of final results The operations are performed in MapReduce, as shown in Section 5.3.3.

The steps require to keep a list of the centers, used by the Map functions. In addition, it is necessary to define some parameters.

Minimum cluster size Any cluster which size is less than is it ignored.

Minimum distance between clusters The algorithm adds a new object to the list of the centers only if its distance from the center is above this parameter.

5.3.1 Addition of the centers

The purpose of the algorithm is first to identify the clusters, then to select the most peripheral element from any one of them. The steps are the following.

1. **Distances computation** For any object, the algorithm computes the distances between the centers and it.

2. **Assignment** The algorithm assigns any object to the nearest center.

3. **Identification of the most distant object** For any cluster, the algorithm computes the distances between its objects and the center.

4. **Identification of the most distant object** For any cluster, the algorithm chooses the object which distance is the highest as the new center, with the condition that the distance is above the fixed threshold.

Steps 1 and step 2 consist in a Map procedure since the computation affects any object separately. As regards steps 3 and step 4, they are about any cluster, so they are performed during the Reduce step.

***MapReduce* 7.** *Addition of the centers.*

Map *The worker computes the distances between any object and the centers. Then, it matches the distances, obtaining the index of the nearest center that is the key. As regards the value, it consists in the object features and in the minimum between the distances.*

Combine *For any key, the worker chooses the object which distance from the center is the highest.*

Reduce *For any key, the master chooses the object which distance from the center is the greatest. If the distance is above the fixed threshold, the output consists in the object features. Otherwise, there is no output.*

The output consists in the list of the new centers.

5.3.2 Removal of the outliers

The purpose of this step is to remove from the list the centers of the small clusters. This is the procedure.

1. **Computation of the distances** For any object, the algorithm computes the distance between the center of its cluster and it.
2. **Assignment** The algorithm assigns any object to the nearest center.
3. **Computation of the size of any cluster** For any cluster, the algorithm counts the number of assigned objects.
4. **Removal of small clusters** The algorithm matches the size of any cluster with the fixed threshold and it removes the center of the cluster if its size is below the threshold.

Similarly to the addition of the centers, the Map step includes step 1 and step 2 while step 3 and step 4 are performed during the Reduce step.

***MapReduce* 8.** *Removal of the outliers.*

Map *For any object, the worker computes the distances between the centers and it. Then, it matches the distances, obtaining the index of the nearest center, and it uses it as the key. The value consists in the object features.*

Combine *For any key, the worker computes the value as the count of the objects associated with it.*

Reduce *If the Combine step has been performed, the master computes the size by adding together the values. Otherwise, in order to do that, it counts the number of objects associated with any key. In both cases, if the size is above the fixed threshold, the output consists in the corresponding center, extracted from the list of the centers. Otherwise, there is no output.*

The output consists in the list of the centers that has not been removed.

5.3.3 Computation of final results

Some possible final results are the size, given by the number of objects belonging to any cluster, the mean, given by the barycenter, and the variance.

The computation of the size of any cluster is rather easy, as it only consists in a count. Each cluster is defined by a list of centers, so any object is assigned

to the cluster that owns the nearest center. The procedure for the computation of any mean is the same.

***MapReduce* 9.** *Computation of the size of any cluster.*

Map *The worker computes the distances between any object and the centers and it matches the distances, identifying the lowest. The key is the label of the nearest center and the value is equal to "1".*

Combine *For any key, the worker sums up the values.*

Reduce *For any key, the master sums up the values.*

The output consists in a list containing the sizes of the clusters.

***MapReduce* 10.** *Computation of the means of the clusters.*

Map *The worker computes the distances between any object and the centers and identifies the lowest. The key is the label of the nearest center and the value consists in the object features.*

Combine *For any key, the worker computes the value as the mean associated with the count of objects.*

Reduce *For any key, the master computes the weighted mean from the weights.*

The output consists in the list of the barycenters of the clusters.

The computation of the variance can be performed using the list of the barycenters. In fact, for any cluster, it is enough to add up the square distances between its object and the barycenter of its cluster.

***MapReduce* 11.** *Computation of the variance of any cluster.*

Map *The worker computes the distances between any object and the centers. The key is the label of the cluster associated with the nearest center and the value is equal to the squared distance between the object and the barycenter of the cluster.*

Combine *For any key, the worker adds up the values and counts the objects. The value consists in the sum and in the counts.*

Reduce *For any key, the master adds up the values and divides them by the sum of the counts (minus one).*

The output consists in the list of the variances of the clusters.

5.4 Split-merge Same-step in MapReduce

Same-step variation of Split-merge algorithm can be implemented in MapReduce, although it requires the use of more computer memory. In fact, since the algorithm needs to keep the labeling of data, it is necessary to copy the entire dataset during each step. Other algorithms need only to store the dataset during the Reduce step, so the total memory requirement is twice its size (six times, considering that HDFS replicates data three times). In this case, instead, the dataset must be stored in another place too, so it is necessary to use three times the dataset size (nine times, due to HDFS replication).

This section shows a MapReduce implementation of the algorithm. As regards the initialization, it simply consists in mapping any data with the key "1", as shown in MapReduce 12. The required parameters are the following.

Centers distance threshold If two centers are too near, the cluster is not split. For this purpose, the split requires that the distance between the identified centers overcomes this threshold.

First cluster size threshold If a cluster size is below this threshold, all its objects are classified as "outliers", so they are removed from the dataset.

Second cluster size threshold, above the first one A cluster that is too big for being classified as "outlier cluster" and too small to overcome this threshold will is the object of an outliers detection. In order to do that, the algorithm splits it into two parts that are a centrally-located one and a peripheral one.

Number of iterations The number of times the algorithm iterates the split procedure.

After the initialization, each step consists in the repeating of som MapReduce jobs, introduced in the following list and described in detail afterwards.

Computation of the size of any cluster Count of the number of objects belonging to any cluster. The implementation is shown in MapReduce 13.

Removal of the outliers Copy of all data, except the ones that belong to small clusters, i.e. to that clusters which sizes are below the first threshold. The implementation is shown in MapReduce 14.

Centers computation Identification of two new centers for any cluster. The implementation is shown in MapReduce 15.

Update of the centers Execution of a single K-means step within any cluster, in order to update the new centers. The implementation is shown in MapReduce 16.

Near centers removal In-memory computation of the distance between any couple of new centers belonging to the same cluster. If it is below the fixed threshold, the cluster will not be split. For this purpose, one of the two centers is removed from the list.

Assignment For any cluster, if there are two new centers, any object is assigned to the nearest. The implementation is shown in MapReduce 17

As regards the merge, it is the same as in the other variant of Split-merge (see Section 5.3).

5.4.1 Initialization

The purpose of this step is to prepare the dataset for the processing. At the beginning, all the objects belong to the same cluster, so they are mapped with the same key.

***MapReduce* 12.** *Map with "1" key.*

Map *The algorithm associates any object with "1" key.*

Combine *The output is the same as the input.*

Reduce *The output is the same as the input.*

The output is the list of data, associated with "1" key each.

5.4.2 Computation of the sizes of the clusters

The purpose of this step is to compute the number of objects belonging to each cluster.

***MapReduce* 13.** *Computation of the sizes of the clusters.*

Map *The output is the same as the input since objects are already mapped with the right key.*

Combine *For any key, computation of the number of the objects of the chunk labeled with it. The value is the count and the key is the same.*

Reduce *For any key, sum up the values.*

The output is a list containing the sizes of the clusters.

5.4.3 Removal of the outliers

The purpose of this step is to create a new dataset that does not contain the objects classified as "outliers". A quicker variant performs the detection during the assignment stage after the centers computation, but it is easier to describe the procedure in this way since it follows the logic flow of the overall algorithm.

***MapReduce* 14.** *Removal of the outliers.*

Map *Any object is given as an output only if its key does not belong to a cluster which size is below the first fixed threshold. The key and the value are the same.*

Combine *Input and output are the same.*

Reduce *Input and output are the same.*

The output consists in the same dataset as the input, without the detected outliers.

5.4.4 Computation of the centers

The purpose of this step is to compute two new centers for each cluster. The computation depends on the cluster size, so each step of the MapReduce algorithm must check the dimensions of the cluster associated with the key. Furthermore, since the centers are two, there are two different outputs for each key. In order to distinguish them, one is mapped with a different key that is equal to the cluster key, plus the number of centers. In this way, it is easy to match the new key with the old one and there are no overlaps between the keys.

***MapReduce* 15.** *Centers computation.*

Map *For any object, there are two possibilities.*

Medium-sized cluster *If the key is matched with a cluster which size is below the second fixed threshold, the value of the output consists in the features of the object and in the distance between the cluster center and it. As regards the key, it is the same.*

Big cluster *Otherwise, the value consists in the object features and in a weight set to "1", in order to have the same length as the output of a medium-sized cluster. The key is randomly chosen from* $\{k, k+k_{\max}\}$*, where* k *is the key of the center and* $k_{\max}$ *is the maximum key before this step.*

Combine *For any object, there are two possibilities.*

Medium-sized cluster *If the key is matched with a cluster which size is below the second threshold, the output consists in two objects. The first value is the object that is the nearest to the center of the cluster associated with the distance between the center and it, with the same key. The other value consists in a mean, associated with the count of objects, with a key that is the old key plus the number of centers.*

Big cluster *Otherwise, the value consists in the mean of the features of the objects and in the count of the number of objects that will be used as a weight. As regards the key, it is the same.*

Reduce *For any object, there are two possibilities.*

Medium-sized cluster *If the key is matched with a cluster which size is below the second threshold, the value consists in the object that is the nearest to the cluster center.*

Big cluster *If the key corresponds to a big cluster, the value consists in the mean and in the count of the number of objects, used as a weight.*

New cluster *If the key is a new one, i.e. it is above the previous maximum key, the value is a weighted mean.*

The output is the list of all the new centers.

5.4.5 Update of the centers

This step performs a single step of K-means clustering within each past cluster. The match of couples of centers is rather easy since the difference between their keys is equal to the past number of centers.

***MapReduce* 16.** *Update of the centers.*

Map *Computation of the distances between any object and the new centers associated with its cluster. The key is the one matched with the nearest center and the value is the same.*

Combine *The value consists in the mean and in the count of the objects.*

Reduce *The value consists in a weighted mean from the counts.*

The output is the updated list of the centers.

5.4.6 Assignment

The purpose of this step is to assign any object to the nearest center between the ones that come from its past cluster.

***MapReduce* 17.** *Assignment to the centers.*

Map *Computation of the distances between any object and the centers corresponding to its cluster. The key corresponds to the nearest center one.*

Combine *Input and output are the same.*

Reduce *Input and output are the same.*

The output is the dataset, with each object mapped with the key of the nearest cluster.

Chapter 6

Testing of the algorithms

In this chapter we describe the testing of the algorithms performances.

At the beginning, the work has consisted in a Proof of concept required by the company and Section 6.1 describes the work and some results. Nevertheless, they are not enough for the evaluation of the algorithms, for the following reasons.

Algorithm Data are clustered through K-means only, so the other algorithms are not handled at all.

Software The analysis has been conducted through Mahout, that is a library that supplies some MapReduce algorithms, such as K-means. On the other hand, the algorithms described in Chapter 5 are implemented through R, linked with Hadoop.

Privacy policies It is not allowed to show the results.

Since the Proof on concept is not enough, the algorithms have been tested on some simulated datasets, as described in Section 6.3. The results are about the computational cost of all the algorithms and they are shown in Section 6.4.

6.1 Real data analysis brief overview

This section briefly describes the analysis on the real customer base data. The work is about a project for a customer of the consulting company where the internship took part. In detail, the data are about the customer base of a big retail chain.

As regards the thesis, the main purpose of this section is to show the performances of K-means since it is the only testing that uses a rather powerful cluster of computers. In addition, this section describes the overall analysis process, in order to show the use of the part of the Hadoop ecosystem described in Section 2.3.

At the beginning, the dataset consists in a structured set of tables, contained in a data warehouse. The main information is a fact table, i.e the set of tickets. In addition, there are some lookup tables that contain some information about the customers and the products.

Since the target of the analysis is the customer base, the first step consists in building a data mart, i.e. a structured set of information about each customer. In detail, each one of them is described by some KPI (Key Performance Indicators), i.e. numeric features that profile the customers on the base of their purchase habits. The KPI computation is based upon the aggregation of the tables. In detail, the chosen fact table consists in the tickets that belong to a chosen period that in this case lasts for three months. As regards the lookup tables, they are the ones related to the fact table. The analysis uses the following 34 KPI.

Monetary The total of expenses.

Frequency The number of ticket markets.

Departments parts The percentage of money spent in items belonging to each one of the 30 departments.

Promo part The percentage of money spent in promotional-offerings items.

Branded part The percentage of money spent in retail-store-branded items.

For example, the computation of each customer monetary consists in the total amount of all his/her tickets imports in the chosen period. In order to compute the KPI, it is necessary to aggregate the ticket table with the one that contains the cost af any item. As regards the dataset size, there are about 1.5 million customers and the tables have a total size of about 180 GB.

The analysis has been conducted through a cloud cluster of 5 computers, with each

RAM: 16 GB;

CPU: quad-core with 2.8 Ghz;

hard-disk: 500 GB.

The computation is performed through 4 computers since the other is the master and it only coordinates the jobs.

The first stage of the data process consists in the import of the DWH data into the Hadoop environment through Sqoop. In order to allow a further process, data are stored in Hive tables. The import takes about 10 hours, but it is not a significant information since an eventual weekly update is much quicker.

The aggregation is performed through 5 Hive scripts, written in HQL. The output of this stage is a data mart that consists in a 5 GB table containing the 34 KPI of all customers. The aggregation takes about 2 hours.

The segmentation of the customer base is conducted using K-means clustering, performed through Mahout libraries. In fact, Mahout includes some MapReduce algorithms, such as K-means. The desired number of clusters is set to 10 and it is the only parameter. As regards the required time, it is about 15 minutes.

6.2 Algorithms testing targets

The purpose of this testing is to give an estimation of the performances of all the clustering algorithms described in Chapter 5. The subject of the analysis is the computational cost that depends both on the dataset size and on the resources, i.e. the calculating capacity of the cluster of computers.

Any described clustering algorithm consists in a sequence of MapReduce jobs, with the addition of some quick in-memory operations. Since the most of any computation consists in some MapReduce steps, the total cost, with a good approximation, is the sum of their costs. Fortunately, it is possible to have an evaluation of any step. As a matter of fact, it depends only on the dataset size and on the computational power. As regards the overall process, its evaluation is more complicated since the required number of steps depends also on the shape of the dataset.

Since the available cluster of computers is not much powerful, the main purpose of the testing is to evaluate how the cost varies between a MapReduce and another. The cluster size is the same for all the testings, so the computational cost depends only on the algorithm and on the dataset size.

The first part of the testing consists in the comparison between different MapReduce algorithms. An important fact is that there are two main classes of

them, on the base of the Reduce features. The first class is about the algorithms that require an expensive Reduce step, such as the computation of a mean, e.g. as regards the K-means update of the centers. The other class contains the algorithms which Reduce operation is quick, such as a count or a sampling, e.g. in the case of K-means random sampling initialization. The difference is that in the first case the cost of both Map and Reduce step grows as the volume increases, while in the second case the increment affects mainly the Map step. The comparison is based on the run of all the algorithms of any same class on the same small-sized dataset and the output is a list of the relative costs.

The second stage of the testing is about the variation of the performances as the size grows. According to the results of the previous testing, it is enough to consider only a single MapReduce algorithm of any class. In fact, starting from that, it is easy to evaluate the performances of the others, using the results of the first step.

The Map step performs the same operations over all data, so its cost is proportional to the dataset size. The Reduce cost behavior, instead, is more complicated since it is an operation between all the objects with the same key. In the case of the second class of MapReduce, it is about the same as dataset grows. On the other hand, if the algorithm belongs to the first class, it is proportional to the size too. In addition to that, the overall cost is increased by some Hadoop needs, such as the management of the jobs and the access to the data. For these reason, there is likely an extra cost. In conclusion, the cost growth is not necessarily linear.

Once the experiments are done, it is possible to have an estimation of the total cost of any algorithm, based only on the number of steps.

6.3 Testing settings

6.3.1 Simulated datasets

In the chosen dataset, we have 5 sets of 10-dimensional data. Any set s is identified by its center c_s, chosen at random.

$$c_s \sim N(0, \sigma_1 I_{10}), \forall s = 1, \dots 5$$

where I_{10} is the 10-dimensional identity matrix.

Any set s contains n_s data that are normal-distributed around the center.

$$x_s^i \sim N(c_s, \sigma_2 I_{10}), \forall i = 1, \dots n_s$$

with $\sigma_2 << \sigma_1$.

There are $n = n_1 + \cdots + n_5$ objects that can belong each to a cluster with probability proportional to $10, 100, 1, 50$, and 20, respectively, so the sizes n_s are different. As regards n, it varies on a wide range and it depends on the testing. In order to simulate the datasets, an R script iteratively generates small chunks and appends them to a csv file. The file size (in Megabytes) is proportional to n:

$$size = 1.7 \times 10^{-4} \times n$$

6.3.2 Computers clusters settings

The execution of algorithms is conducted through an Amazon cloud cluster of computers. Each node is an Amazon small instance and it has the following setup.

RAM 1.7 GB.

CPU 1 virtual core with 1 EC2 Compute Unit (1 GHz).

As regards the storage, it is variable, according to the necessities.

Since the computational power of the cluster is not much high, the maximum processable volume of data is rather low. The problem is that HDFS chunks have a default size of 64 MB that overcomes the size of the used dataset, so the algorithms would not be parallelized. For that reason, the HDFS chunk size has been set to lower values, in order to always have at least 4 chunks of data. In this way, each one of the four worker nodes has at least a chunk to process.

6.3.3 Classes of MapReduce algorithms

The first class of MapReduce algorithms is about the ones that have an expensive, although scalable, Reduce, such as a mean or a sum. The algorithms are in the following list.

K-means random partitioning initialization Reduce is a mean.

K-means step Reduce is a mean.

K-means++ single sampling Reduce is a weighted sampling.

K-means++ psi computation Reduce is a sum.

Split-merge addition of the centers Reduce is a maximum.

Split-merge update of the centers Reduce is a mean.

One-step Split-merge addition of the centers Reduce is a mean.

One-step Split-merge update of the centers Reduce is a mean.

The second class of MapReduce algorithms is about the algorithms which Reduce does not become more expensive as the dataset size increases. For instance, it can be a sampling, or a carrying of data, i.e. an absence of Reduce. The algorithms are in the following list.

K-means random sampling initialization Reduce is a sampling.

K-means++ multiple sampling Reduce is a bind between matrices.

Split-merge removal of the outliers Reduce does not perform any operation.

One-step Split-merge size Reduce is a count.

One-step Split-merge outliers Reduce does not perform any operation.

One-step Split-merge assignment Reduce does not perform any operation.

6.4 Results

6.4.1 Single steps comparison

The comparison between the single steps has been conducted on a small-sized dataset, with 2×10^4 data and a size of 3.4 MB. As regards the HDFS blocks size, it is set to 1 MB. The R code used to assess the required time is shown in Appendix B.2. The following table illustrates the results.

Algorithm	**Cost (sec)**	**Class**
K-means random partitioning initialization	92.855	1
K-means step	401.362	1
K-means++ single sampling	415.458	1

K-means++ psi computation	267.873	1
Split-merge addition of the centers	389.678	1
Split-merge update of the centers	392.568	1
One-step Split-merge addition of the centers	90.989	1
One-step Split-merge update of the centers	91.767	1
K-means random sampling initialization	79.926	2
K-means++ multiple sampling	273.939	2
Split-merge Removal of the outliers	388.017	2
One-step Split-merge size	89.242	2
One-step Split-merge outliers	88.992	2
One-step Split-merge assignment	105.532	2

The variation of the computational cost is tested only on K-means step and K-means random sampling initialization, so the important information is the relative cost, i.e. the rate between the cost of any step and of the chosen algorithm of the class. Since the data about the required time are not exact, the relative costs are approximate. Let us call t_1 the cost of a K-means step and t_2 the cost of a K-means random sampling initialization. The following table shows the relative costs of all MapReduce jobs.

Algorithm	**Cost (sec)**
K-means partitioning initialization	1/3 t_1
K-means step	1 t_1
K-means++ single sampling	1 t_1
K-means++ psi computation	2/3 t_1
Split-merge addition of the centers	1 t_1
Split-merge update of the centers	1 t_1
One-step Split-merge addition of the centers	1/3 t_1
One-step Split-merge update of the centers	1/3 t_1
K-means random sampling initialization	1 t_2
K-means++ multiple sampling	10/3 t_2
Split-merge removal of the outliers	5 t_2
One-step Split-merge size	1 t_2
One-step Split-merge outliers	1 t_2
One-step Split-merge assignment	4/3 t_2

6.4.2 Computational cost of two single steps

The range of the dataset sizes is between 2×10^4 and 20×10^4. The reason is that the cost is more than an hour if the size overcomes the threshold. The step is of 2×10^4 and it is constant, in order to have a good estimation of the trend. As regards the dataset chunk size, it is chosen in such a way that there are always 4 blocks since there are 4 worker nodes. In fact, the volume is too small for needing the use of more blocks. The following table reports the features of the datasets.

Number of data	**File size (MB)**	**Block size (MB)**
2.00E+4	3.4	1
4.00E+4	6.8	2
6.00E+4	10.2	3
8.00E+4	13.6	4
1.00E+5	17	5
1.20E+5	20.4	6
1.40E+5	23.8	7
1.60E+5	27.2	8
1.80E+5	30.6	9
2.00E+5	34	10

As shown in Figure 6.1, the cost of a K-means step grows linearly with the increment of the volume. As regards the slope, it is about $0.0158\,sec/datum$. Conversely, as shown in Figure 6.2, the growth of the cost of K-means random sampling initialization is more problematic. In fact, the growth is slower and is less predictable. We can say that the cost is constant and it is about $100\,sec$.

Fortunately, these performances are about a small-sized cluster, so they are not explanatory about the real ones. Nevertheless, they may help in the calculation of the cost of a bigger cluster.

6.4.3 Computational cost of the overall algorithms

Before estimating the cost of the algorithms, let us give a list of the required MapReduce steps. Let S be the number of steps and K the desired number of centers. The steps and the cost t of any complete clustering algorithm are in the following list.

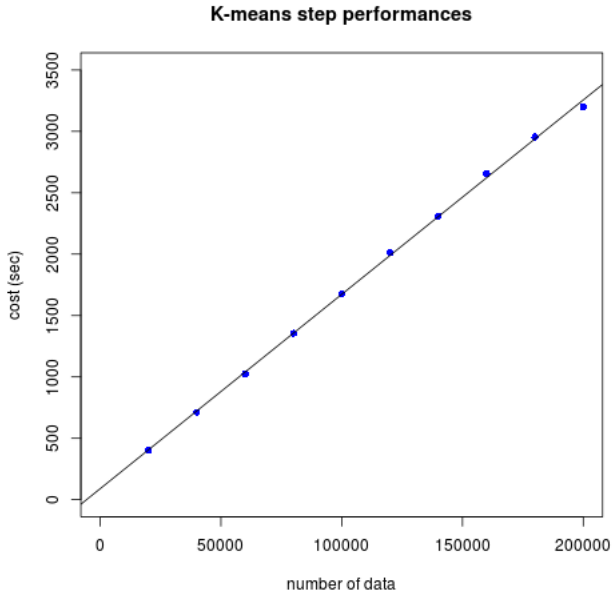

Figure 6.1: Cost growth of a K-means step.

K-means K-means initialization (random partitioning or random sampling), then S K-means steps.

Random partitioning

$$t = \frac{1}{3}t_1 + St_1 = \left(\frac{1}{3} + S\right)t_1$$

Random sampling

$$t = t_2 + St_1$$

K-means++ K K-means++ single sampling.

$$t = Kt_1$$

Scalable K-means++ $S + 1$ psi computations and S multiple K-means++ samplings.

$$t = \frac{2(S+1)}{3}t_1 + \frac{10S}{3}t_2$$

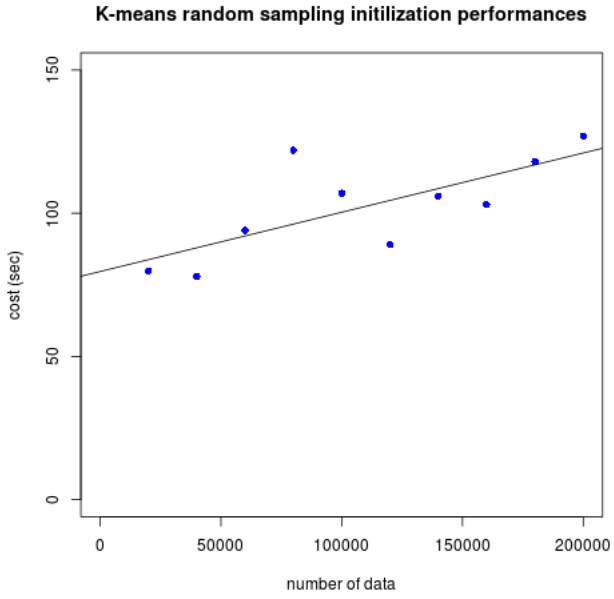

Figure 6.2: Cost growth of a K-means random sampling initialization.

Split-merge S repetitions of the three split steps, i.e. addition of the centers, update of the centers, and outliers detection. The initial step, i.e. K-means, is not included.

$$t = S\left(t_1 + t_1 + 5t_2\right) = 2St_1 + 5St_2$$

One-step Split-merge S repetitions of the four split steps, i.e. computation of the sizes of the clusters, outliers detection, addition of the centers, update of the centers, and assignment.

$$t = S\left(t_2 + t_2 + \frac{1}{3}t_1 + \frac{1}{3}t_1 + \frac{4}{3}t_2\right) = \frac{2S}{3}t_1 + \frac{10S}{3}t_2$$

Chapter 7

Conclusions

This chapter summarizes and evaluates the overall work. Section 7.1 discusses the results, whereas Section 7.2 concerns some possible continuations of the work.

7.1 Thesis overview

In this thesis we have presented some techniques that allow the handle of big volumes of data through cluster analysis. Since all the methods are based on MapReduce, the core of the work is the development of this kind of algorithms.

As regard K-means, there is nothing new in its MapReduce implementation. As a matter of fact, there are several tools that provide it, since it is one of the most popular clustering algorithms. By the way, its re-implementation through R-Hadoop has allowed us to gain familiarity with MapReduce. As regards K-means initialization, the tools we know provide only the random sampling and the random partitioning. Scalable K-means++, instead, is described in [15]. However, since we have not found any software that provides it, its implementation was necessary.

The innovative part of this work consists in the development of new Hybrid clustering techniques. In fact, they combine a slow handle of big volumes of data with a fast in-memory process. The idea is to use the job parallelization only when it is needed, in order to have more effective results. The validity of this kind of solution is confirmed by the existence of similar algorithms in literature (see [16]). By the way, our opinion is that these hybrid clustering techniques

may still be significantly improved and that the main lack is a criterion for the choice of the parameters. In conclusion, we think that they could potentially be the starting point for other algorithms.

As regards the techniques applications, there are different fields, such as Customer Relationship Management and fraud detection. The algorithms are scalable, so there is no limit in the maximum volume of processable data, despite the maximum amount of available resources. Furthermore, the use of a big cluster of computers allows a very quick process, so the methods can provide quick answers. As regards the cost, the use of Hadoop keeps it rather low, since the hardware does not have any particular requirements. For all these reasons, the techniques fit well some Big Data problems.

7.2 Future developments

As mentioned in Section 1.1, Big Data is about the problems affected by the "Three V", i.e. volume, velocity, and variety. Unfortunately, this paper deals mainly with the volume problem and only partly with the velocity. As regards the variety, it does not take part at all in the work, despite we consider it as the more fascinating challenge.

Some examples of unstructured data are lists, graphs, and texts. Since the starting point of cluster analysis is the distance between two objects, first it is necessary to define how to compute it. As regards lists, the distance can be a value that is inversely proportional to the number of elements in common. Also the case of texts is similar, since it is possible to convert a text in a wordcount list. As regards graphs with weighted edges, the distance between two objects is inversely proportional to the weight of the edge that links them.

Once the distance is defined, it is possible to perform hierarchical clustering since the algorithms are only based upon it. By the way, this solution is not scalable over big volumes of data, so it is necessary to use centroid-based clustering. However, in this way, the algorithm requires the computation of centroids, that is impossible in the case of some kinds of unstructured data. For this purpose, there are some alternatives. For example, in the case of graphs, it is possible to define a "generalized median graph"[17], instead of a barycenter.

In conclusion, this paper deals with only a small part of Big Data cluster analysis and there is a multitude of possible continuations.

Appendix A

R-Hadoop Codes

This appendix shows the codes used to implement the MapReduce algorithms. The used software is R, connected with Hadoop through "rmr2" package, introduced in Subsection 2.4.1. Each clustering algorithm recalls some MapReduce subroutines, that are functions that follow the model described in Section A.1. Since the difference between two MapReduce jobs affects only the Map and Reduce function, the section reports them only.

A.1 Mapreduce functions

This is the model for any MapReduce job.

```
mrStep = function(
  pathin,
  pathout,
  input.format = 'native',
  output.format = 'native'
){
  cat('\n\nName of the step\n\n')
  if(hdfs.exists(pathout)) hdfs.rm(pathout)

  mapFunction = function(k, v) {
      # computation of key and val
      keyval(key, val)
    }
```

```
  comFunction = function(k, v) {
      # computation of key and val
      keyval(key, val)
    }

  redFunction = function(k, v) {
      # computation of key and val
      keyval(key, val)
    }

  mapreduce(
    input = pathin,
    output = pathout,
    input.format = input.format,
    output.format = output.format,
    map = mapFunction,
    combine = comFunction,
    reduce = redFunction
  )
}
```

A.2 K-means

A.2.1 K-means main

```
mrKmeans = function(
  pathin,
  pathout,
  parKmeans,
  clustersSize = T
){
  cat('\n\nKMEANS',
      '\nnumber of iterations: ', parKmeans$nIter, '\n\n')

  if(is.null(parKmeans$init))
    parKmeans$init = 'sample'
```

```
# 1) INITIALIZATION
if(is.null(parKmeans$centers)){
  if(is.null(parKmeans$nCenters)){
    cat('\nWARINING: number of centers set to default (2)')
    parKmeans$nCenters = 2
  }
  cat('\nCENTERS INITIALIZATION\n')
  if(parKmeans$init == 'random' | parKmeans$init == 1)
    mrKmeansInitRandom(pathin, pathout)
  else if(parKmeans$init == 'sample' | parKmeans$init == 2)
    mrKmeansInitSample(pathin, pathout)
  else if(parKmeans$init == 'kmeans++ scalable' | parKmeans$init == 4)
    mrKmeansppScalable(pathin, pathout, parKmeans=parKmeans, nIter=2)
  else if(parKmeans$init == 'kmeans++ one step' | parKmeans$init == 5)
    mrKmeanspp(pathin, pathout, parKmeans)
  else{
    cat('\nERROR: no initialization method')
    return()
  }

  if(is.null(parKmeans$centers)){
    centers = from.dfs(pathout)$val
    centers = matrixOrder(centers)
    colnames(centers) = paste('V', 1:dim(centers)[2], sep = '')
    parKmeans$centers = centers
    cat('\ncenters:\n')
    print(centers)
  }
}
else{
  cat('\ncenters are already initialized')
}
parKmeans$centersList = list(centers0=parKmeans$centers)

# 2) ITERATIVE UPDATE
for(i in 1:parKmeans$nIter){
```

```
    cat('\nK MEANS ITERATION', i)
    mrKmeansStep(pathin, pathout)
    output = from.dfs(pathout)$val
    centers = matrixOrder(output)
    colnames(centers) = paste('V', 1:dim(centers)[2], sep = '')

    if(dim(centers)[1] == dim(parKmeans$centers)[1])
      if(prod(centers == parKmeans$centers)){
        cat('\nsame centers as the previous step!',
            '\nnew centers:\n')
        print(centers)
        cat('\nold centers:\n')
        print(parKmeans$centers)
        mrKmClusterSize(pathin, pathout)
        if(clustersSize)
          parKmeans$clustersSize = from.dfs(pathout)$val
        return(parKmeans$centers)
      }else
        cat('\ncenters changed')

    parKmeans$centers = centers
    parKmeans$centersList[[paste('centers',i,sep='')]] = centers
    cat('\ncenters:\n')
    print(centers)
  }
  mrKmClusterSize(pathin, pathout)
  if(clustersSize)
    parKmeans$clustersSize = from.dfs(pathout)$val

  parKmeans$centers
}
```

A.2.2 K-means parameters initialization

```
mrParKmeans = function(
  nCenters=NULL,
  centers=NULL,
```

```
  distFun=NULL
){
  print('# K-means parameters initialization')
  if(is.null(distFun))
    distFun = function(centers, data){
      # warning: it is the square of the distance!
      if(is.null(dim(centers))) {
        distances = array(NA, c(1, dim(data)[1]))
        for(j in 1:length(distances))
          distances[1, j] = sum((centers - data[j, ]) ^ 2)
        return(distances)
      }
      if(dim(centers)[2] == 1) centers = t(centers)
      distances = array(NA, c(dim(centers)[1], dim(data)[1]))
      for(i in 1:dim(centers)[1])
        for(j in 1:dim(data)[1])
          distances[i, j] = sum((centers[i, ] - data[j, ]) ^ 2)
      return(distances)
    }
  parList = new.env(parent=globalenv())
  class(parList) = 'pointer'
  parList$nCenters = nCenters
  parList$centers = centers
  parList$distFun = distFun
  parList
}
```

A.2.3 K-means step

mrKmeansStep

```
mapFunction = function(k, v) {
distances = parKmeans$distFun(parKmeans$centers, v)
key = as.matrix(max.col(-t(distances)))
keyval(key, v)
}

redFunction = function(k, v){
```

```
val = t(as.matrix(colMeans(v)))
colnames(val) = paste('V', 1:dim(val)[2], sep = '')
keyval(k, val)
}
```

A.2.4 K-means random split initialization

mrKmeansInitRandom

```
mapFunction = function(k, v) {
key = as.matrix(sample(1:parKmeans$nCenters, nrow(v), replace=T))
keyval(key, v)
}

redFunction = function(k, v){
val = t(as.matrix(colMeans(v)))
colnames(val) = paste('V', 1:dim(val)[2], sep = '')
keyval(k, val)
}
```

A.2.5 K-means random sample initialization

mrKmeansInitSample

```
mapFunction = function(k, v) {
key = as.matrix(sample(1:parKmeans$nCenters, nrow(v), replace=T))
keyval(key, rbind(v))
}

redFunction = function(k, v){
sampledId = sample(1:dim(v)[1], 1)
val = rbind(v[sampledId, ])
#     key = as.matrix(k[1])
key = 1
keyval(k, val)
}
```

A.2.6 K-means++ initialization

K-means++ initialization main

```
mrKmeanspp = function(
  pathin,
  pathout,
  parKmeans
){
  cat('\n\nKMEANS++ (NORMAL)\n\n')
  for(i in 1:parKmeans$nCenters){
    cat('\nKmeans++ iteration', i, '\n')
    mrKmeansppStep(pathin, pathout)
    newCenter = from.dfs(pathout)$val
    parKmeans$centers = rbind(parKmeans$centers, newCenter)
    colnames(parKmeans$centers) = paste('V', 1:dim(parKmeans$centers)[2], sep='')
    rownames(parKmeans$centers) = paste('center', 1:dim(parKmeans$centers)[1], sep='')
    cat('\ncenters:\n')
    print(parKmeans$centers)
  }
  cat('\n kmeanspp end \nsampled points:\n')
  return(parKmeans$centers)
}
```

K-means++ step

```
mrKmeanspp = function(
  pathin,
  pathout,
  parKmeans
){
  cat('\n\nKMEANS++ (NORMAL)\n\n')
  for(i in 1:parKmeans$nCenters){
    cat('\nKmeans++ iteration', i, '\n')
    mrKmeansppStep(pathin, pathout)
    newCenter = from.dfs(pathout)$val
    parKmeans$centers = rbind(parKmeans$centers, newCenter)
    colnames(parKmeans$centers) = paste('V', 1:dim(parKmeans$centers)[2], sep='')
```

```
    rownames(parKmeans$centers) = paste('center', 1:dim(parKmeans$centers)[1], sep='')
    cat('\ncenters:\n')
    print(parKmeans$centers)
  }
  cat('\n kmeanspp end \nsampled points:\n')
  return(parKmeans$centers)
}
```

A.2.7 Alternative K-means++ initialization

K-means++ initialization main

```
mrKmeansppScalable = function(
  pathin,
  pathout,
  parKmeans,
  l = 10, # it is about the number of centers sampled in each iteration
  nIter = NULL,
  maxcentersNumber = 1000
){
  library(LICORS)

  cat('\n\nK-means ++ SCALABLE\n\n')
  mrSampleKmeansppFirst(pathin,pathout)
  parKmeans$centerspp = from.dfs(pathout)$val
  colnames(parKmeans$centerspp) = paste('V', 1:dim(parKmeans$centerspp)[2], sep='')
  rownames(parKmeans$centerspp) = 'center1'
  centersNumber = 1

  cat('\n\npsi initial computation\n\n')
  mrMinDistSquareSum(pathin,pathout)
  parKmeans$psi = from.dfs(pathout)$val
  cat('\n\npsi = ', parKmeans$psi,'\n\n')

  if(is.null(nIter))
    nIter=ceiling(log(parKmeans$psi))
  cat('\n\nK MEANS++ NUMBER OF ITERATIONS: ', nIter, '\n\n')
  for(i in 1:nIter) if(parKmeans$psi>0){
```

```
    cat('\n\nK MEANS++ ITERATION', i, '\n\n')
    mrSampleKmeanspp(pathin,pathout)
    newCenters = from.dfs(pathout)$val

    if(length(newCenters) > 1){
      maxcentersNumberAvailable = maxcentersNumber - centersNumber
      if(is.null(dim(newCenters)))
        newcentersNumber = 1
      else
        newcentersNumber = dim(newCenters)[1]
      if(newcentersNumber > maxcentersNumberAvailable){
        print('too many new centers: only some of them will be used')
        newCenterChosen = sample(1:newcentersNumber, maxcentersNumberAvailable)
        newCenters = newCenters[newCenterChosen, ]
      }

      parKmeans$centerspp = rbind(parKmeans$centerspp, newCenters)
      parKmeans$centerspp = matrixDuplicatesElimination(parKmeans$centerspp)
      rownames(parKmeans$centerspp) = paste('center', 1:dim(parKmeans$centerspp)[1], sep='')


      mrMinDistSquareSum(pathin,pathout)
      parKmeans$psi = from.dfs(pathout)$val
      if(parKmeans$psi == 0){
        cat('\n\nWARNING: psi=0\n\n')
        return(parKmeans$centerspp)
      }
      centersNumber = dim(parKmeans$centerspp)[1]
#       if(centersNumber >= maxcentersNumber)
#         return(parKmeans$centerspp)
    }
    cat('\n\ncenters:\n')
    print(parKmeans$centerspp)
    cat('\n\npsi = ', parKmeans$psi,'\n\n')
  }
  if(is.null(dim(parKmeans$centerspp)))
    cat('\n\nERROR: KMEANS++ (SCALABLE) HAS NOT SAMPLED ANY POINT\n\n')
```

```
  if(dim(parKmeans$centerspp)[1] < parKmeans$nCenters)
    cat('\n\nERROR: KMEANS++ (SCALABLE) HAS NOT SAMPLED ENOUGH POINT\n\n')

#   mrClusterSizepp(pathin,pathout)
#   cat('\nkmeans++ (scalable) end\nsample size:\n', dim(parKmeans$centerspp)[1], '\n')
  centers = kmeanspp(parKmeans$centerspp, parKmeans$nCenters)$centers
  centers = kmeans(parKmeans$centerspp, centers)$centers
  parKmeans$centers = centers
  cat('\n\nfinal K-means++ centers:\n')
  print(parKmeans$centers)


  return(centers)
}
```

Sum of minimum square distances

mrMinDistSquareSum

```
mapFunction = function(k, v){
distances = parKmeans$distFun(parKmeans$centerspp, v)
val = as.matrix(apply(distances, 2, min))
key = matrix(1, dim(val)[1])
return(keyval(key, val))
}

redFunction = function(k, v){
keyval(1, sum(v))
}
```

K-means++ initial sample

mrSampleKmeansppFirst

```
mapFunction = function(k, v){
if(is.null(dim(v))){
  val = rbind(v)
  return(keyval(1, val))
}
```

```
sampledId = sample(1:dim(v)[1], 1)
val = v[sampledId, ]
keyval(1, val)
}

redFunction = function(k, v){
if(is.null(dim(v))){
  val = rbind(v)
  return(keyval(k, val))
}
sampledId = sample(1:dim(v)[1], 1)
val = v[sampledId, ]
keyval(1, val)
}
```

K-means++ sample

mrSampleKmeanspp

```
mapFunction = function(k, v){
val = matrix(1, 0, dim(v)[2])
for(i in 1:dim(v)[1]){
  distances = parKmeans$distFun(parKmeans$centerspp, rbind(v[i, ]))
  d = min(distances)
  prob = parKmeans$l * d / parKmeans$psi
  if(runif(1,0,1) < prob) val = rbind(val, v[i, ])
}
if(dim(val)[1] == 0){
  return(keyval(0, 0))
  l$log = paste(l$log, '\n WARNING in function mrSampleKmeanspp')
}

if(dim(val)[1] == 0)
  return(NULL)

key = matrix(1, dim(val)[1])
return(keyval(key, val))
}
```

```
redFunction = function(k, v){
keyval(k, v)
}
```

A.3 Split-merge

A.3.1 Split-merge main

```
smMain = function(
  pathin,
  pathout,
  parSm,
  parKmeans,
  outputHead = T
  ){
  # INITIALIZATION
  if(is.null(parKmeans$nCenters))
    parKmeans$nCenters = 4
  if(is.null(parKmeans$nIter))
    parKmeans$nIter = 4
  if(is.null(parKmeans$init))
    parKmeans$init = 'sample'
  if(is.null(parSm$minDist))
    parSm$minDist = 0.5

  smInitMain(pathin, pathout, parKmeans)


  centers = parKmeans$centers
  cat('\n\n REMOVAL OF NEAR CENTERS')
  cat('\n\n centers:\n')
  print(centers)
  nCenters = dim(centers)[1]
  if(nCenters > 1){
    removeId = nCenters + 1
    for(i in 2:nCenters)
```

```
      for(j in 1:(i - 1))
        if(sum((centers[i, ] - centers[j, ]) ^ 2) < parSm$minDist)
          removeId = c(removeId, i)
    centers = centers[-removeId, ]
  }
  cat('\n\n CENTERS AFTER THE REMOVAL')
  cat('\n\n centers:\n')
  print(centers)


  parSm$centers = centers

  # SPLIT
  if(is.null(parSm$nIter))
    parSm$nIter = 2
  if(is.null(parSm$distFun))
    parSm$distFun = parKmeans$distFun
  if(is.null(parSm$minClusterSize))
    parSm$minClusterSize = 5

  smSplitMain(pathin, pathout, parSm)


  # MERGE
  if(is.null(parSm$linkage))
    parSm$linkage = 'single'
  if(is.null(parSm$cutMethod))
    parSm$cutMethod = 'number' # or height
  if(is.null(parSm$cutHeight))
    parSm$cutHeight = 1.2
  if(is.null(parSm$cutNumber))
    parSm$cutNumber = 4

  smMergeMain(pathin, pathout, parSm)

  return(pathout)
}
```

A.3.2 Split-merge parameters initialization

```
mrParSm = function(
  distFun=NULL,
  centers=NULL
){
  cat('\n# split&merge parameters initialization')
  if(is.null(distFun))
    distFun = function(centers, data){
      # warning: it is the square of the distance!
      if(is.null(dim(centers))) {
        distances = array(NA, c(1, dim(data)[1]))
        for(j in 1:length(distances))
          distances[1, j] = sum((centers - data[j, ]) ^ 2)
        return(distances)
      }
      if(dim(centers)[2] == 1) centers = t(centers)
      distances = array(NA, c(dim(centers)[1], dim(data)[1]))
      for(i in 1:dim(centers)[1])
        for(j in 1:dim(data)[1])
          distances[i, j] = sum((centers[i, ] - data[j, ]) ^ 2)
      return(distances)
    }
  parList = new.env(parent=globalenv())
  class(parList) = 'pointer'
  parList$centers = centers
  parList$distFun = distFun
  parList
}
```

A.3.3 Split initialization

```
smInitMain = function(
  pathin,
  pathout,
  parKmeans
){
  cat('\n\nSPLIT-MERGE INITIALIZATION\n\n')
```

```
  mrKmeans(pathin, pathout, parKmeans)
  cat('\n\nsplit-merge initialization end\n\n')
  return(pathout)
}
```

A.3.4 Split

Split main

```
smSplitMain = function(
  pathin,
  pathout,
  parSm
){
  cat('\n\nSPLIT-MERGE ITERATIVE SPLITTING\n\n')
  parSm$removedCenters = NULL

  for(i in 1:parSm$nIter){
    cat('\n\nSPLIT-MERGE SPLITTING ITERATION ', i, '\n\n')
    mrSmSplitAdd(pathin, pathout)
    newCenters = from.dfs(pathout)$val
    if(is.matrix(newCenters))
      parSm$centers = rbind(parSm$centers, newCenters)
    rownames(parSm$centers) = NULL
    cat('\n\ncenters after add ', i, '\n')
    print(parSm$centers)

    mrSmSplitUpdate(pathin, pathout)
    parSm$centers = from.dfs(pathout)$val
    rownames(parSm$centers) = NULL
    cat('\n\ncenters after update ', i, '\n')
    print(parSm$centers)

    mrSmSplitClean(pathin, pathout)
    parSm$centers = from.dfs(pathout)$val
    cat('\n\ncenters after clean ', i, '\n')
    print(parSm$centers)
  }
```

```
  cat('\n\nsplit-merge iterative splitting end\n\n')

  return(pathout)
}
```

Centers addition

mrSmSplitAdd

```
  mapFunction = function(k, v) {
    distances = parSm$distFun(parSm$centers, v)
    key = as.matrix(max.col(-t(distances)))
    val = cbind(v, apply(t(distances), 1, min))
    keyval(key, val)
  }

  redFunction = function(k, v){
    chosenID = which.max(v[, dim(v)[2]])
    dist = max(v[, dim(v)[2]])
    key = as.matrix(k[1])
    val = v[chosenID, 1:(dim(v)[2] - 1)]
    val = rbind(val)
    #     if(sum((val - parSm$centers[key]) ^ 2) < parSm$minDist ^ 2)
    #       return(NULL)
    if(dist < parSm$minDist)
      return(NULL)
    return(keyval(key, val))
  }
```

Update of the centers

mrSmSplitUpdate

```
mapFunction = function(k, v) {
distances = parSm$distFun(parSm$centers, v)
key = as.matrix(max.col(-t(distances)))
keyval(key, v)
}
```

```
redFunction = function(k, v){
val = t(as.matrix(colMeans(v)))
keyval(k, val)
}
```

Outliers detection

mrSmSplitClean

```
mapFunction = function(k, v) {
distances = parSm$distFun(parSm$centers, v)
key = as.matrix(max.col(-t(distances)))
keyval(key, v)
}

redFunction = function(k, v) {
clusterSize = dim(v)[1]
if(clusterSize < parSm$minClusterSize){
  return(NULL)
}else{
  val = rbind(parSm$centers[k, ])
  return(keyval(k, val))
}
}
```

A.3.5 Merge

Merge main

```
smMergeMain = function(
  pathin,
  pathout,
  parSm
){
  cat('\n\nSPLIT-MERGE MERGE\n\n')

  d = dist(parSm$centers)
  clusts <- hclust(d, method=parSm$linkage)
  if(parSm$cutMethod == 'number'){
```

```
    k <- cutree(clusts, k = parSm$cutNumber)
  }else if(parSm$cutMethod == 'height'){
    k <- cutree(clusts, h = parSm$cutHeight)
  }else{
    cat('\n\nERROR: no dendogram cut method!!!\n\n')
  }

  mrSmMergeCount(pathin, pathout)
  dimensionsSplit = from.dfs(pathout)$val

  dimensionsMerged = list()
  cat('\n\nMERGED CLUSTERS DIMENSIONS:\n')
  for (key in unique(k)){
    dimensionsMerged[[key]] = sum(dimensionsSplit[k == key])
    cat('\n cluster ', key, ': ', dimensionsMerged[[key]], '\n')
  }
  parSm$dimensionsMerged = dimensionsMerged

  return(pathout)
}
```

Merge count

mrSmMergeCount

```
mapFunction = function(k, v) {
distances = parSm$distFun(parKmeans$centers, v)
key = as.matrix(max.col(-t(distances)))
keyval(key, v)
}

redFunction = function(k, v){
val = as.matrix(dim(v)[1])
keyval(k, val)
}
```

A.4 One-step Split-merge

A.4.1 One-step Split-merge main

```
sm1Main = function(
  pathin,
  pathout,
  pathtemp,
  parSm,
  parKmeans,
  outputHead = T
  ){
  # INITIALIZATION
  if(is.null(parKmeans$nCenters))
    parKmeans$nCenters = 4
  if(is.null(parKmeans$nIter))
    parKmeans$nIter = 10
  if(is.null(parKmeans$init))
    parKmeans$init = 'sample'
  if(is.null(parSm$minDist))
    parSm$minDist = 0.5

  sm1InitMain(pathin, pathout, pathtemp)
  parSm$centers = from.dfs(pathout)$val

  # SPLIT
  if(is.null(parSm$nIter))
    parSm$nIter = 2
  if(is.null(parSm$distFun))
    parSm$distFun = parKmeans$distFun
  if(is.null(parSm$minClusterSize))
    parSm$minClusterSize = 5

  sm1SplitMain(pathtemp, pathout, pathin, parSm)


  # MERGE
```

```
  if(is.null(parSm$linkage))
    parSm$linkage = 'single'
  if(is.null(parSm$cutMethod))
    parSm$cutMethod = 'number' # or height
  if(is.null(parSm$cutHeight))
    parSm$cutHeight = 1.2
  if(is.null(parSm$cutNumber))
    parSm$cutNumber = 4

  smMergeMain(pathtemp, pathout, parSm)

  return(pathout)
}
```

A.4.2 One-step Split-merge initialization

Initialization main

```
sm1InitMain = function(
  pathin,
  pathout,
  pathtemp
){
  cat('\n\nSPLIT-MERGE ONE-STEP INITIALIZATION\n\n')
  mrSm1InitLabel(pathin, pathtemp)
  mrSm1InitMean(pathin, pathout)
  cat('\n\nsplit-merge initialization end\n\n')
  return(pathout)
}
```

Initial labeling

mrSm1InitLabel

```
mapFunction = function(k, v) {
key = matrix(1, dim(v)[1])
val = rbind(v)
return(keyval(key, val) )
}
```

```
redFunction = function(k, v){
return(keyval(k, v))
}
```

Computation of the barycenter

mrSm1InitMean

```
mapFunction = function(k, v) {
key = 1
val = rbind(colMeans(v))
val = cbind(val, dim(v)[1])
keyval(key, val)
}

redFunction = function(k, v){
key = 1
val = matrix(0, 1, dim(v)[2] - 1)
for(i in 1:dim(v)[1]){
  val = val + rbind(v[i, 1:(dim(v)[2] - 1)]) * v[i, dim(v)[2]]
}
val = val / sum(v[, dim(v)[2]])
return(keyval(key, val))
}
```

A.4.3 One-step Split

One-step Split main

```
sm1SplitMain = function(
  pathin,
  pathout,
  pathtemp,
  parSm
){
  cat('\n\nSPLIT-MERGE ONE-STEP ITERATIVE SPLITTING\n\n')

  printCenters = function(centers){
```

```
    cat('\n')
    for(i in 1:length(centers))
      if(!is.null((centers[[i]]))){
        cat('\ncenter ', i, ': ', sep='')
        cat(centers[[i]])
      }
    cat('\n\n')
  }

  firstCenter = parSm$centers
  parSm$centers = list()
  parSm$centers[[1]] = firstCenter

  for(iter in 1:parSm$nIter){
    cat('\n\nSPLIT ITERATION ', iter, '\n\n')

    mrSm1SplitSize(pathin, pathout)
    output = from.dfs(pathout)
    parSm$size = array(0, max(output$key))
    for(i in 1:length(output$key)){
      clusterId = output$key[i]
      parSm$size[clusterId] = output$val[i]
    }

    cat('\n\nclusters sizes at the beginning of step ', iter, ' :\n')
    print(parSm$size)

    mrSm1SplitOutliers(pathin, pathtemp)

    cat('\n\ncenters before add ', iter, ' :\n')
    printCenters(parSm$centers)

    parSm$maxKey = 0
    for(i in 1:length(parSm$centers))
      if(!is.null(parSm$centers[[i]]))
        parSm$maxKey = i
```

```
mrSm1SplitAdd(pathtemp, pathout)
output = from.dfs(pathout)

parSm$centers = list()
for(i in 1:(parSm$maxKey * 2))
  parSm$centers[[i]] = NA
for(iCenter in 1:dim(output$val)[1])
  parSm$centers[[output$key[iCenter]]] = output$val[iCenter, ]

cat('\n\ncenters after add ', iter, ' :\n')
printCenters(parSm$centers)
cat('\n\nmax key:\n')
print(parSm$maxKey)

mrSm1SplitUpdate(pathtemp, pathout)
output = from.dfs(pathout)

parSm$centers = list()
for(i in 1:(parSm$maxKey * 2))
  parSm$centers[[i]] = NA
for(iCenter in 1:dim(output$val)[1])
  parSm$centers[[output$key[iCenter]]] = output$val[iCenter, ]


cat('\n\ncenters after update ', iter, ' :\n')
printCenters(parSm$centers)

cat('\n\nnear centers removal at step ', iter, ' :\n')

for(i in 1:parSm$maxKey)
  if(prod(is.na(parSm$centers[[i + parSm$maxKey]])) == 0 &
       prod(is.na(parSm$centers[[i]])) == 0)
    if(sum((parSm$centers[[i]] - parSm$centers[[i + parSm$maxKey]]) ^ 2)
       < parSm$minDist){
      parSm$centers[[i + parSm$maxKey]] = NA
    }
cat('\n\nclusters to split at step', iter, ' :\n')
```

```
    for(i in 1:parSm$maxKey)
      if(prod(is.na(parSm$centers[[i + parSm$maxKey]])) == 0)
        cat(' ', i)
    cat('\n')

    mrSm1SplitAssign(pathtemp, pathin)

  }
  cat('\n\nsplit-merge one-step iterative splitting end\n\n')
  centers = NULL
  cat('\n\nfinal centers: \n')
  for(i in 1:length(parSm$centers))
    if(sum(is.na(parSm$centers[[i]])) == 0){
      print(parSm$centers[[i]])
      centers = rbind(centers, parSm$centers[[i]])
  }
  parSm$centers = centers

  return(pathout)
}
```

Clusters sizes computation

```
mapFunction = function(k, v) {
key = k
val = 1
keyval(key, val)
}

redFunction = function(k, v){
key = k[1]
val = sum(v)
return(keyval(key, val))
}
```

Outliers detection

mrSm1SplitOutliers

```
mapFunction = function(k, v) {
return(keyval(k, v))
}

redFunction = function(k, v){
key = k[1]
if(parSm$size[key] < parSm$minClusterSize1)
  return(NULL)

return(keyval(k, v))
}
```

Centers addition

mrSm1SplitAdd

```
mapFunction = function(k, v) {
key = k
val = matrix(NA, dim(v)[1], dim(v)[2] + 1)
for(i in 1:dim(v)[1]){
  if(parSm$size[k[i]] < parSm$minClusterSize2){
    distance = sum((rbind(v[i, ]) - parSm$centers[[k[i]]]) ^ 2)
    if(is.na(distance))
      distance = Inf
    val[i, ] = cbind(rbind(v[i, ]), distance)
  }else{
    val[i, ] = cbind(rbind(v[i, ]), 1)
    key[i] = k[i] + sample(c(0, parSm$maxKey), 1)
  }
}
for(i in 1:length(parSm$centers))
  if(parSm$size[k[i]] < parSm$minClusterSize2)
    if(prod(is.na(parSm$centers[[i]])) == 0)
      val = rbind(val, cbind(rbind(parSm$centers[[i]]), 1))
keyval(key, val)
}

redFunction = function(k, v){
```

```
key = k[1]
val = rbind(colMeans(v[, 1:(dim(v)[2] - 1)]))
return(keyval(key, val))
}
```

Update of the centers

mrSm1SplitUpdate

```
mapFunction = function(k, v) {
key = k
val = v
for(i in 1:dim(v)[1]){
  if(prod(is.na(parSm$centers[[k[i] + parSm$maxKey]])) > 0){
    key[i] = key[i] + parSm$maxKey
  }else{
    distance1 = sum((parSm$centers[[k[i]]] - v[i, ]) ^ 2)
    distance2 = sum((parSm$centers[[k[i] + parSm$maxKey]] - v[i, ]) ^ 2)
    if(distance1 < distance2)
      key[i] = key[i] + parSm$maxKey
  }
}
keyval(key, v)
}

redFunction = function(k, v){
key = k[1]
val = t(as.matrix(colMeans(v, na.rm = T)))
if(sum(is.na(val)) > 0){
  val = matrix(Inf, 1, dim(v)[2])
}
keyval(key, val)
}
```

Assignment to the centers

mrSm1SplitAssign

```
mapFunction = function(k, v) {
```

```
key = k
for(i in 1:dim(v)[1]){
  if(prod(is.na(parSm$centers[[k[i] + parSm$maxKey]])) == 0 &
       prod(is.na(parSm$centers[[k[i]]])) == 0){
    distance1 = sum((parSm$centers[[k[i]]] - v[i, ]) ^ 2)
    distance2 = sum((parSm$centers[[k[i] + parSm$maxKey]] - v[i, ]) ^ 2)
    if(distance1 < distance2)
      key[i] = key[i] + parSm$maxKey
  }
}
keyval(key, v)
}

redFunction = function(k, v){
return(keyval(k, v))
}
```

Appendix B

R Codes

In this chapter, we report some R codes used.

B.1 testing datasets

This section presents the R codes that generated the testing datasets.

B.1.1 Semicircle dataset

```
testSemicircle = function(n){
  n1 = n/2
  n2 = n/2
  rho1 = runif(n1, 0, 1)
  rho2 = rnorm(n2, 5, 0.2)
  theta1 = runif(n1, 0, 2*pi)
  theta2 = runif(n2, 0, pi)
  x1 = cbind(rho1 * sin(theta1), rho1 * cos(theta1))
  x2 = cbind(rho2 * sin(theta2), rho2 * cos(theta2))
  x = rbind(x1, x2)
  return(x)
}
```

B.1.2 Dartboard dataset

```
testDartboard = function(n){
```

```
  n1 = n2 = n3 = ceiling(n/3)
  rho1 = rep(0, n1) + rnorm(n1, 0, 0.1)
  rho2 = rep(2, n2) + rnorm(n2, 0, 0.1)
  rho3 = rep(4, n3) + rnorm(n3, 0, 0.1)
  theta1 = seq(0, 2*pi, length.out=n1)
  theta2 = seq(0, 2*pi, length.out=n2)
  theta3 = seq(0, 2*pi, length.out=n3)
  x1 = cbind(rho1 * sin(theta1), rho1 * cos(theta1))
  x2 = cbind(rho2 * sin(theta2), rho2 * cos(theta2))
  x3 = cbind(rho3 * sin(theta3), rho3 * cos(theta3))
  x = rbind(x1, x2, x3)
  return(x)
}
```

B.1.3 Mouse dataset

```
testMouse = function(n){
  n1 = n*0.75
  n2 = n3 = (n-n1)/2
  rho1 = runif(n1, 0, 4.5)
  rho2 = runif(n2, 0, 2)
  rho3 = runif(n3, 0, 2)
  theta1 = runif(n1, 0, 2*pi)
  theta2 = runif(n2, 0, 2*pi)
  theta3 = runif(n3, 0, 2*pi)
  x1 = cbind(rho1 * sin(theta1), rho1 * cos(theta1))
  x2 = cbind(rho2 * sin(theta2) + 5, rho2 * cos(theta2)+ 5)
  x3 = cbind(rho3 * sin(theta3) - 5, rho3 * cos(theta3)+ 5)
  x = rbind(x1, x2, x3)
  return(x)
}
```

B.1.4 Spiral dataset

```
testSpiral = function(n){
  n1 = n2 = n/2
  rho1 = seq(0, 5, length.out=n1) + runif(n1, 0, 0.1)
  rho2 = seq(1, 6, length.out=n2) + runif(n1, 0, 0.1)
```

```
  theta1 = seq(0, 4*pi, length.out=n1)
  theta2 = seq(0, 4*pi, length.out=n2)
  x1 = cbind(rho1 * sin(theta1), rho1 * cos(theta1))
  x2 = cbind(rho2 * sin(theta2), rho2 * cos(theta2))
  x = rbind(x1, x2)
  return(x)
}
```

B.1.5 Four clusters dataset

```
testFourDistinct = function(n){
  n1 = n2 = n3 = n4 = ceiling(n/4)
  rho1 = rep(0, n1) + rnorm(n1, 0, 1)
  rho2 = rep(0, n2) + rnorm(n2, 0, 1)
  rho3 = rep(0, n3) + rnorm(n3, 0, 1)
  rho4 = rep(0, n4) + rnorm(n4, 0, 1)
  theta1 = seq(0, 2*pi, length.out=n1)
  theta2 = seq(0, 2*pi, length.out=n2)
  theta3 = seq(0, 2*pi, length.out=n3)
  theta4 = seq(0, 2*pi, length.out=n4)
  x1 = cbind(rho1 * sin(theta1) + 5, rho1 * cos(theta1) + 5)
  x2 = cbind(rho2 * sin(theta2) + 5, rho2 * cos(theta2) - 5)
  x3 = cbind(rho3 * sin(theta3) - 5, rho3 * cos(theta3) + 5)
  x4 = cbind(rho4 * sin(theta4) - 5, rho4 * cos(theta4) - 5)
  x = rbind(x1, x2, x3, x4)
  return(x)
}
```

B.1.6 Add noise function

```
addNoise = function(x, noiseRate = 0.01, varScale = 0.2){
  functionName = 'addNoise'
  if(!is.matrix(x)) cat(verboseInputClass(functionName, 'x', 'matrix'))
  if(!is.numeric(noiseRate)) cat(verboseInputClass(functionName, 'N', 'noiseRate'))

  if(noiseRate == 0) return(x)
  n = ceiling(dim(x)[1] * noiseRate)
  var = sqrt(sum(max.col(x) + max.col(-x))) / dim(x)[2] * varScale
```

```
  xNew = NULL
  for(j in 1:n){
    xNewSingle = NULL
    for(p in 1:dim(x)[2]){
      xNewSingle = c(xNewSingle, rnorm(1, 0, var))
    }
    xNew = rbind(xNew, xNewSingle)
  }
  x = rbind(x, xNew)
  if(!is.matrix(x)) cat(verboseOutputClass(functionName, 'x', 'matrix'))
  return(x)
}
```

B.2 Computational cost

This small piece of R code has been used to asses the computational cost of some algorithms.

```
time = proc.time()
# algorithm
cat('\n\ntime for the algorithm: ',
  proc.time()[3] - time[3], 'seconds\n\n')
```

Bibliography

[1] Chris Eaton, Dirk DeRoos, Tom Deutsch, George Lapis, Paul Zikopoulos (2012): *Understanding Big Data*, IBM

[2] Douglas, Laney (2012): *The Importance of "Big Data": A Definition*, Gartner

[3] Dean J., Ghemawat, S. (2004): *MapReduce: Simplified Data Processing on Large Clusters*, Proc of the Sixth Symposium on Operating Systems Design and Implementation (OSDI 2004), San Francisco, California

[4] Gantz J., Reinsel D. (2010): *The Digital Universe Decade – Are You Ready?*, EMC

[5] White T. (2012): *Hadoop: the definitive guide*, O'Reilly Media

[6] Borthakur, D. (2007): *The Hadoop Distributed File System: Architecture and Design*, Hadoop Project Website

[7] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, R. Murthy (2011): *Hive: a warehousing solution over a map-reduce framework* , Proceedings of the VLDB Endowment

[8] Lars G. (2011): *HBase: the definitive guide*, O'Reilly Media

[9] Chang F., Dean J., Ghemawat S., Hsieh W. C., Wallach D. A., Burrows M., Chandra T., Fikes A., Gruber R. E. (2006): *Bigtable: A Distributed Storage System for Structured Data*, ACM Transactions on Computer Systems

[10] (2012): *Advanced 'Big Data' Analytics with R and Hadoop*, Revolution Analytics

[11] MacQueen J.B. (1967): *Some Methods for Classification and Analysis of MultiVariate Observations*, Proc. of the fifth Berkeley Symposium on Mathematical Statistics and Probability, 1, p. 281-297. University of California Press

[12] Pena J.M., Lozano J.A., Larranaga P. (1998): *An empirical comparison of four initialization methods for the K-means algorithm*, Proceedings of the fifth Berkeley Symposium on Mathematical Statistics and Probability, 1, p. 281-297. University of California Press

[13] Forgy E.W. (1965) *Cluster analysis of multivariate data : efficiency versus interpretability of classifications* Biometrics, Vol. 21, p. 768-780

[14] Arthur D., Vassilvitskii S. (2007) *K-means++: The Advantages of Careful Seeding* Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms, p. 1027-1035. Society for Industrial and Applied Mathematics Philadelphia, PA, USA

[15] Bahmani B., Moseley B., Andrea Vattani A., Ravi Kumar R., Vassilvitskii S. (2012) *Scalable K-means++* Proceedings of the VLDB Endowment, p. 622-633. VLDB Endowment, USA

[16] Murugesan K., Zhang J. (2011) *Hybrid hierarchical clustering: an experimental analysis* Department of Computer Science, University of Kentucky, Lexington (USA)

[17] Ferrer M., Valveny E., Serratosa F., Bardaj I., Bunke H. (2009) *Graph-based k-means Clustering: A Comparison of the Set Median versus the Generalized Median Graph* Lecture Notes in Computer Science Volume 5702, p. 342-350. Springer.