Computational Science and Engineering

Technische Universität München

Master's Thesis

Scalable Stream Clustering in Apache Spark

Omar Backhoff Larrazolo

Computational Science and Engineering

Technische Universität München

Master's Thesis

Scalable Stream Clustering in Apache Spark

Skalierbares Clustern von Stream-Daten in Apache Spark

Author:	Omar Backhoff Larrazolo
1st examiner:	Prof. Dr. Michael Gerndt
2nd examiner:	Prof. Dr. Hans-Joachim Bungartz
Assistant adviser:	Prof. Dr. Eirini Ntoutsi
Thesis handed in on:	2nd of May, 2016

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

April 29, 2016 Omar Backhoff Larrazolo

Acknowledgments

I would not have enough words to describe how much I appreciate every single one who helped me be in the place I am. Many thanks to Technical University of Munich and the Department of Informatics, the Leibniz Supercomputing Center, my examiners and adviser, my friends, family and particularly my parents. Without the support of everyone involved in this process, this thesis could not have been possible.

Abstract

Two of the most popular strategies to mine big data are distributed computing and stream mining. The purpose of this thesis is to incorporate both together bringing a competitive stream clustering method into a modern framework for distributed computing, namely, Apache Spark. The method in question is *CluStream*, a stream clustering method which separates the clustering process into two different phases: an online phase which handles the incoming stream, generating statistical summaries of the data and an offline phase which takes those summaries to generate the final clusters. These summaries also contain valuable information which can be used for further analysis. The main goal is to adapt this method in such a framework in order to obtain a scalable stream clustering method which is open source and can be used by the Apache Spark community.

Contents

List of Figures

1. Introduction

The analysis of data streams comes along with important questions: what kind of data is it? What important information is contained in it? How does the stream evolve? The key question for this project among those is the latter, i.e. dealing with the evolution of the stream, because prior to the development of the *CluStream*[8] method there was not an easy to answer that question as it was one of the first to tackle this issue.

There are two main types of data for clustering: static and streams. The former is perhaps the most common and there are plenty of good and well known algorithms for that, such as K-Means and Mixture of Gaussians[9]. Normally, static data is not expected to change over time and it is already given in a single database (or dataset). On the contrary, in a stream environment the data is constantly arriving (being streamed) and it might change over time; the details of this are covered in the chapter 2. Although the first category is out of the scope of this work, it is important to mention that it is widely used in many situations; when one does not have access to the source of the data or when (real-) time analysis is not required. Additionally, some of the main reasons data would be streamed instead of stored are because it is either too big or it is of little use and just the most important information is needed. Some examples of data streams would be: social networks feeds, information captured by sensors constantly monitoring a phenomenon and network traffic.

Clustering is one of the main tasks in data mining, also often referred as an exploratory subtask of it. As the name implies, the objective is to find clusters, i.e., collections of objects that share common properties. One can also relate this task to unsupervised machine learning, which intends to classify data when it lacks of

labels, i.e., when the data instance does not indicate to which category it belongs.

The *CluStream* method was developed in 2003[8] and its main purpose is to provide more information than previously developed algorithms for data stream clustering by that time. It provides a solution for handling streams of data independently from the one that finds the final clusters. It consists of two phases (passes) instead of one; the first one deals with the incoming data and stores relevant information over time and the second one is in charge of the clustering using the previously generated information. In other words,

1. For each batch of data, statistically relevant "summaries" of the data are created and stored at a defined pace. This storing pace follows a specific storage scheme such that the disk space requirement reduces drastically; this is necessary as in most cases for data streams one does not want to store everything that arrives, one reason being the big data requires large and expensive computational resources (processing power and storage).

2. On user demand, the stored "summaries" can be used for the end clustering task as they include all necessary information to achieve accurate results. Additionally, as these summaries are stored over time, a user defined time horizon/window can be chosen in order to analyze the data in different time periods, giving the possibility of a better understanding of the evolution of the data.

1.1. Context of this work

This thesis was done is in cooperation with the *Ludwig-Maximilians-Universität (LMU)*, Munich. It is under direct advising of Prof. Dr. Eirini Ntoutsi from the Department of Informatics in the Faculty of Mathematics, Informatics and Statistics of the mentioned university.

She is a data scientist in the areas of Data Mining, Machine Learning, Information Retrieval and Databases with research interests such as pattern extraction,

change detection, evolution monitoring and reasoning over complex dynamic data / data streams.

As of March 2016 she joined the Faculty of Electrical Engineering and Computer Science at the *Lebniz Universität Hannover* as an Associate Professor. Supervision was mainly done while with LMU.

1.2. Goals and relevance of this work

In short, the first goal is to bring the *CluStream* method to the world of distributed computing, in particular to the *Apache Spark* framework, which is a newer approach to distributed computing. By October 2015 there are no implementations of this algorithm for this framework, i.e. it is not part of the core code of *Spark* and there are no public external packages available in the *Spark-packages.org* collection.

As part of the implementation process, performance tuning and optimization criteria have to be taken into consideration in order to obtain not only the availability of the algorithm in the framework but to have it taking full advantage of the power of distributed computing, and that is the second goal.

Having those two goals accomplished, the code will be available as open source code in *GitHub* and as an external library in *Spark-packages.org*, which allows anyone to easily incorporate it on existing builds of *Spark*.

Why it matters

According to a survey realized by a company called *Typesafe*[2], realized to more than 2000 developers, data scientists and C-level executives who work with related tools, by the time it was released:

- 31% were evaluating *Spark*
- 20% were planning to use it in 2015
- 13% were already using it in production

- 82% of the current users replace *MapReduce* with it
- 78% of the current users need faster processing for larger datasets
- 67% of the current users plan to introduce event stream processing

These are some of the numbers that indicate that this framework is gaining a lot of support and its adoption increases rapidly. Given its importance nowadays it seems coherent to contribute to this project, to give developers and scientists the option of using modern stream clustering tools in a modern framework.

Besides its growth in popularity, *Spark* is a framework that contains different modules, "all in one". More specifically, it includes libraries of machine learning algorithms, for stream processing, an SQL module to work with structured data and graph/graph-parallel computation[1], with the aim of simplifying the workflow of big data analysis, making it a clear choice for this project.

1.3. Roadmap

1.3.1. Background and theory

In the first part of this work, the necessary background to understand the problematic better is detailed; this include an introduction to the programming model *MapReduce* for distributed computing and a discussion of important clustering algorithms, including *Streaming K-Means*.

Following the background comes the theory of the two most relevant concepts for this work, which are *Apache Spark* and the *CluStream* algorithm. Regarding *Spark* there are three points of interest that are described: what it is and how parallel computations can be performed, its streaming platform and relevant tools included. As for *CluStream*, the description of the algorithm is shown along with a discussion on how it can be used for a better understanding of the evolution of data streams.

1.3.2. Implementation

The second part includes the details of the implementation of *CluStream* in *Spark*. For this section the code is presented and explained such that it can be understood how the parallelization of the algorithm was accomplished with the tools provided by the framework.

The implementation consists of two main tasks and a minor one. The first two are the online micro clustering and the offline macro clustering phases of the algorithm; these are the main components of *CluStream* and are the core of this thesis. The minor task is how to actually use the code here implemented from its source or as an external package of *Spark* and would serve as guide for anyone who intends to use it independently of having the knowledge of how it was programmed.

This section also discusses about which methodologies and considerations were used to achieve the best performance regarding speed and scalability -the accuracy of it is strongly inherited from the algorithm itself more than it has to do with the implementation.

1.3.3. Analysis and results

After the implementation comes the analysis of what was achieved. In this third part, the implementation is put into test with different types of data streams to validate its correctness and to measure its performance. Two environments will be used as testing scenarios, these are *Spark* as a standalone installation on a single machine and an installation on a cluster with several processing units, the first one is detailed in Section 3.1 and the second one in chapter 2. The principal reason to do this is to acknowledge its capabilities and limitations.

Additionally, a comparison analysis is presented against the single stream clustering algorithm available by default as part of the core of *Spark*: *Streaming K-Means* and some of the related work. The point of interest is to determine how it stands up against it in terms of performance and scalability showing well grounded advantages and disadvantages when picking one over the other.

1.4. Implementation notes

It is important to point out that this algorithm was not originally designed to have a parallel implementation and therefore this work does not aim to replicate purely all of its aspects but rather to achieve a correct adaptation of it for the given *Spark* framework. Due to the framework's nature, some parts of the algorithm had to be adapted; processing the data in batches instead of single points, this is discussed in chapter 3.

2. Background and Theory

Traditionally, the two main approaches to deal with big data are streaming algorithms and distributed computing[17]. Streaming algorithms process the data in batches and therefore the computational requirements reduce significantly while distributed computing provides additional computational power by dividing a task among multiple processing units.

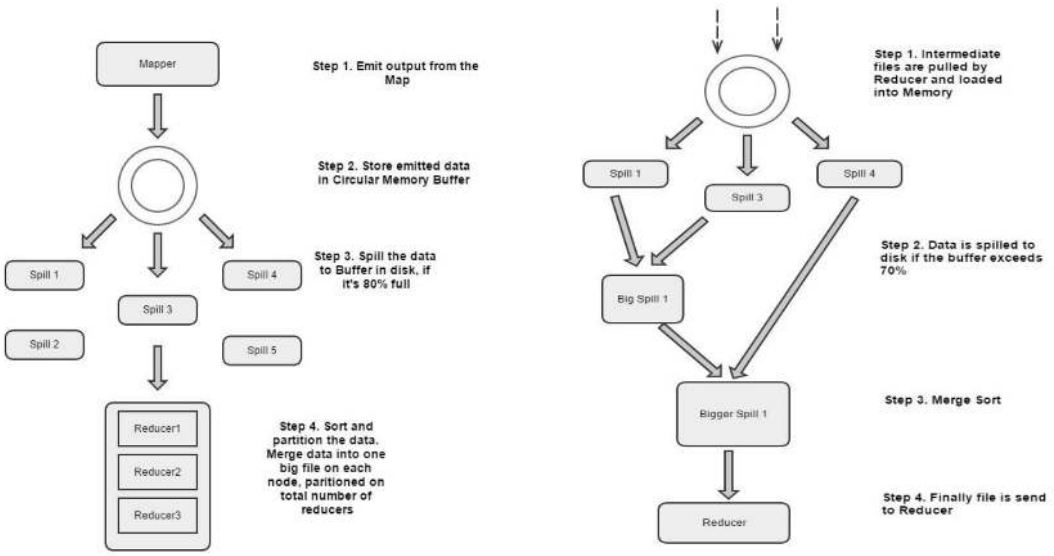
The specific problem that is relevant in this work is stream clustering, for which several streaming algorithms exist, such as Spark's *Streaming K-Means*[5] and *CluStream*.

When it comes to distributed computing, a widely used approach is to use distributed file systems with the *MapReduce* programming model such as *Hadoop* which was inspired by work developed by *Google Inc*. For that reason, the idea behind this model is described in the following subsection. In terms of clustering, and in particular stream clustering, many people have taken this path to develop algorithms that work on distributed file systems to deal with big data and the description of the already mentioned streaming version of K-means is also described here.

2.1. Distributed computing

2.1.1. The MapReduce approach for distributed computing

"MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce



(a) Map phase (b) Reduce phase

Figure 2.1.: The two MapReduce phases[14]

function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model...", as it was stated in the paper published by *Google Inc.* in 2004[11].

This approach has been a very popular option for computational scientists to work with big data in a distributed manner. Clarifying the paragraph above, one first starts with a distributed file system -in their case it was the *Google File system*[13]- where the data is located. Then the programmer has to write two functions: map and reduce; such that they generate key/value pairs in a way that it is possible to transform/operate the values identified by their keys (map), then these intermediate results can be combined by an operation (reduce).

The *Hello World* for MapReduce is a word count program for one or multiple text documents. Hereby an example, provided in the same paper, is shown with algorithms 1 and 2.

Algorithm 1 map(String key, String value)

Input: The *key* is the document name and the *value* is the document contents.
Output: The *key* is a word and the *value* is a "1".

1: **for each** $word = w \in input.value$ **do**
2: $\quad EmitIntermediate(w, "1")$
3: **end for**

Algorithm 2 reduce(String key, Iterator values)

Input: The *key* is a word and the *value* is a list of counts.
Output: The *key* is a word and the *value* is a "1".

1: $int\ result = 0$
2: **for each** $value = v \in input.value$ **do**
3: $\quad result+ = ParseInt(v)$
4: **end for**
5: $Emit(AsString(result))$

2.1.2. Apache Spark

Apache Spark is an open source framework developed in the AMPLab at the University of California, campus Berkeley[3]. It is a fast and general engine for large-scale data processing. The original goal was to design a new programming model that supports a wider class of applications than MapReduce and at the same time to keep the fault tolerance property of it. They claim MapReduce is inefficient for applications that require a multi-pass implementation and a low latency data sharing across parallel operations, which are common in data analytics nowadays, such as:

- Iterative algorithms: many machine learning and graph algorithms.
- Interactive data mining: multiple queries on data loaded into RAM.
- Streaming applications: some require an aggregate state over time.

Traditionally, MapReduce and DAG engines are based on an acyclic data flow, which makes them not optimal for these applications listed above. In this flow,

data has to be read from a stable storage system, like a distributed file system, and then processed on a series of jobs only to be written back to the stable storage. This process of reading and writing data on each step of the workflow causes a significant rise in computational cost.

Spark proposes *resilient distributed datasets (RDDs)* to overcome this issue efficiently. RDDs are stored in memory between queries (no need of replication) and they can rebuild themselves in case of failure as they remember how they were originally built from other datasets by transformations such as *map*, *group*, *join*.

The framework

One of the points where Spark stands out is its generality. Spark provides a series of tools ready to be used and combined, making it suitable for a broad range of applications.

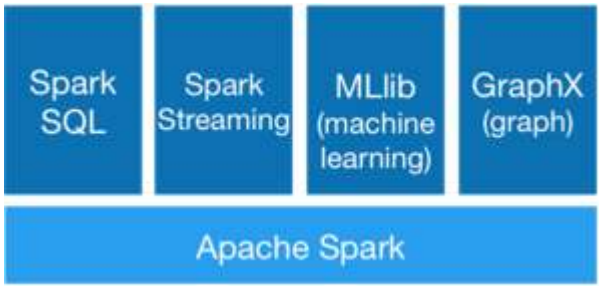


Figure 2.2.: The Spark framework[1]

Figure 2.2 shows the four main components of Spark, which can all interact with one another seamlessly.

- Spark SQL: a module that handles structured data such as JSON, JDBC, Hive,etc. through SQL queries, which are very well known from relational databases.
- Spark Streaming: a module that allows Spark to work with streams of data. These streams can arrive through different means such as sockets, files or custom receivers.

- MLlib: a mature machine learning module built for Spark.
- GraphX: a module to perform graphs and graph-parallel computations.

Within the reach of this work only two modules are of direct interest: Spark streaming and MLlib.

Spark's core

The fundamental idea behind Spark is the concept of *resilient distributed datasets (RDDs)*. Spark relies on RDDs to handle all the operations needed on distributed data.

RDDs are immutable collections of objects partitioned throughout the nodes of a cluster and they are preferably stored in RAM when it's available. They are built through lazy parallel transformations[1] of the data allowing them to be reconstructed on failure.

There are two types of operations that can be applied on RDDs:

- Transformations: these operations are lazy and only the instructions are stored without computing the actual results of them allowing Spark to operate more efficiently, e.g. the computation of all the lazy transformations will occur once they are needed (once an action is requested) such that only the result of that action is returned, rather than a larger dataset containing the transformations. Examples of transformations are: *map, filter and groupBy.*
- Actions: these operations trigger the computation of the transformations assigned to the RDD and execute a final operation. Examples of actions are: *count, reduce and collect.*

Spark streaming

For this project, Spark streaming plays an important role as it takes a raw data stream and transforms it so that it is possible to process it within the framework.

[1] Lazy transformations are those which do not perform computations.

A raw stream of data can come in different forms and through different channels: from a very simple file stream, where whenever a new file is added to a specific location it is recognized as a new instance to a socket stream where the data comes through the network using a TCP protocol. More elaborate stream sources such as *Kafka, Flume, Twitter, HDFS/S3*, among others are also supported.

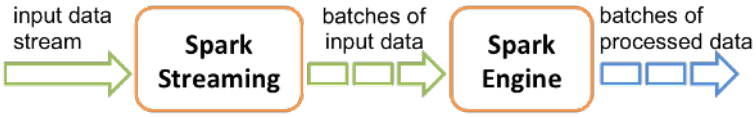


Figure 2.3.: Flow of data in Spark streaming

Figure 2.3 shows the general idea of Spark streaming[4], a raw stream is linked to this module and it converts it to batches of data at user-defined intervals. These batches of data are then treated as RDDs, thus it gets distributed over the cluster where Spark runs. The abstraction of a data stream in Spark is called *DStream*, which stands for Discretized Stream, and is a continuous series of RDDs. In a *DStream*, each RDD contains data from a specific interval of time, as it can be seen in Figure 2.4.

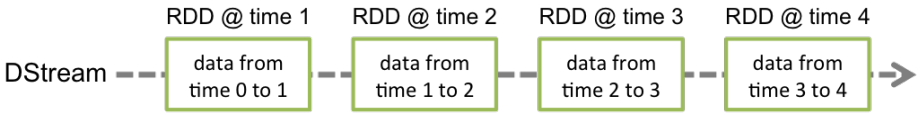


Figure 2.4.: DStreams are Spark streaming's abstraction of a data stream

2.2. Stream clustering

2.2.1. Streaming K-Means

This algorithm is strongly related to regular *K-Means*, described in the appendix A.1 with a slight modification.

The clusters should be able to adapt over time in a way that the clusters are not highly dominated by old data, the reason is that a new batch of data being clustered in a stream would have very little influence on the cluster if all the previous data is used as it is small in proportion. For this reason, a different approach is preferred.

Streaming K-Means is a generalization of the update rule of a method called *Mini Batch K-Means (MBK means)*. *MBK means* is a modification to standard *K-Means* with a faster convergence and delivers only slightly worse results[6]. However, for this method the important thing to consider is how the clusters are updated with a new batch of data: the data points are assigned to their nearest cluster, the new cluster centers are computed and then each cluster is updated as follows[7]:

$$c_{t+1} = \frac{c_t n_t \alpha + x_t m_t}{n_t \alpha + m_t} \tag{2.1}$$

$$n_{t+1} = n_t + m_t \tag{2.2}$$

where c_t is the previous center for the cluster, n_t is the previous number of points assigned to the cluster, x_t is the new cluster center from the current batch of data and m_t is the number of points of current batch of data assigned to the cluster. The parameter α is what controls the decay of old data, e.g. when $\alpha = 1$ all data will be used and when $\alpha = 0$ only new data will be used.

The implementation of this method is then quite similar to normal *K-Means* with the main difference being that it is performed a multiple times as the data batches arrive, updating the centers on each time. The **distributed computing** implementation for this algorithm is similar to the one for *K-Means* and a *MapReduce* approach can be taken, again (c.f. Appendix A.1).

2.2.2. CluStream

CluStream[8] is a method developed in the Watson Research Center at IBM and the University of Illinois, UIUC. This method presented a different approach on stream clustering with respect to a modified versions of *K-Means* which have been adapted to cluster data streams, e.g., *Streaming K-Means* and others[15][12]. The main difference relies on splitting the clustering process into two parts: one which would handle the data stream itself gathering only statistically relevant information (online part) and another which actually processes the results of the former to produce the actual clusters wanted (offline part).

Separating the clustering process provides the user with several advantages, among others:

- by saving only statistical data, rather than the original content, it is possible to save physical storage space (e.g. hard drive space) and therefore reducing costs and allowing a wider range of time to be clustered.
- The method also allows the analysis of the evolution of the data, as the necessary information for that is contained in the stored statistical information.
- Because the two parts operate independently it allows the user to select a time horizon, or even a time window, to perform the offline clustering part using the stored statistical information.

The CluStream framework

This method is built over a few ideas that need to be conceptualized, which will answer fundamental questions and set up a basis of terminology useful along this work.

- **Micro-Clusters**: the statistical information summaries that are computed during the online component. They comprise a temporal extension of *cluster feature vectors*[18], which benefit from the additive property that makes them a natural choice for the data stream problem[8].

- **Pyramidal time frame**: micro-clusters are stored periodically following a pyramidal time-frame. This allows a nice trade-off between the ability to store large amounts of information while giving the user the possibility to work with different time horizons without loosing too much precision. The statistical summaries stored are used by the offline component to compute finally the macro-clusters which are the actual clusters the user intended to get.

It is assumed that a data stream comes in the form of multi-dimensional records $\bar{X}_1 ... \bar{X}_k ...$ where $\bar{X}_i = (x_i^1 ... x_i^d)$.

Definition 2.1. *[8]*

A micro-cluster for a set of d-dimensional points $X_{i_1} ... X_{i_n}$ with time stamps $T_{i_1} ... T_{i_n}$ is defined as the $(2 \cdot d + 3)$ tuple $(\overline{CF2^x}, \overline{CF1^x}, CF2^t, CF1^t, n)$, wherein $\overline{CF2^x}$ and $\overline{CF1^x}$ each correspond to a vector of d entries. The definition of each of these entries is as follows:

- *For each dimension, the sum of the squares of the data values is maintained in $\overline{CF2^x}$. Thus, $\overline{CF2^x}$ contains d values. The p-th entry of $\overline{CF2^x}$ is equal to $\sum_{j=1}^{n} (x_{i_j}^p)^2$.*
- *For each dimension, the sum of the data values is maintained in $\overline{CF1^x}$. Thus, $\overline{CF1^x}$ contains d values. The p-th entry of $\overline{CF1^x}$ is equal to $\sum_{j=1}^{n} x_{i_j}^p$.*
- *The sum of the squares of the time stamps $T_{i_1} ... T_{i_n}$ is maintained in $CF2^t$.*
- *The sum of the time stamps $T_{i_1} ... T_{i_n}$ is maintained in $CF1^t$.*
- *The number of data points is maintained in n.*

The idea behind the pyramidal time frame is that *snapshots* of the micro-clusters can be stored in an efficient way, such that if t_c is the current clock time and h is the history length, it is still possible to find an accurate approximation of the higher level clusters (macro-clusters) for a user specified time horizon $(t_c - h, t_c)$.

In this time frame, the snapshots are stored at different levels of granularity that depends upon their *recency*. These are classified in different orders which can vary from 1 to $log(T)$, where T is the clock time elapsed since the beginning of the stream. The snapshots are maintained as follows:

- A snapshot of the i-th order occurs at time intervals α^i, for $\alpha \geq 1$. In other words, a snapshot occurs when $T \bmod \alpha^i = 0$.
- At any given moment, only the last $\alpha^l + 1$ snapshot for any given order are stored, where $l \geq 1$ is a modifier which increases the accuracy of the time horizon at the cost of storing more snapshots. This allows redundancy, but from an implementation point of view only one snapshot must be kept.
- For a data stream, the maximum number of snapshots stored is $(\alpha^l + 1) \cdot log_\alpha(T)$.
- For any specified time window h, it is possible to find at least one stored snapshot within $2 \cdot h$ units of the current time[2].
- The time horizon h can be approximated to a factor of $1 + (1/\alpha^{l-1})$, whose second summand is also referred as the accuracy of the time horizon.

Order of Snapshots	Clock Times (Last 5 Snapshots)
0	55 ~~54~~ 53 ~~52~~ 51
1	54 ~~52~~ 50 ~~48~~ 46
2	52 ~~48~~ 44 ~~40~~ 36
3	~~48~~ 40 ~~32~~ 24 ~~16~~
4	48 ~~32~~ 16
5	32

Figure 2.5.: Example of snapshots stored for $\alpha = 2$ and $l = 2$

With the help of Figure 2.5 it is possible to observe how this pyramidal time frame works: snapshots of order 0 occur at odd time units, these need to be retained as they are non-redundant. Snapshots of order 1 which occur at time units not divisible by 4 are non-redundant and must be retained. In general, all the snapshots of order i which are not divisible by α^{i+1} are non-redundant. Another thing

[2]The proof can be found in the original article[8].

to note is that whenever a new snapshot of a particular order is stored, the oldest one from that order needs to be deleted.

To illustrate the effect on the accuracy of storing more snapshots, the following example is given: supposing that a stream is running for 100 years, with a time granularity of 1 second. The total number of snapshots stored would be $(2^2 + 1) \cdot log_\alpha(100 * 365 * 24 * 60 * 60) \approx 158$ with an accuracy of $1/2^{2-1} = 0.5$ or 50% of a given time horizon h. Increasing the modifier l to 10 would yield up to $(2^{10} + 1) \cdot log_\alpha(100 * 365 * 24 * 60 * 60) \approx 32343$ maximum snapshots stored with an accuracy of $1/2^{10-1} \approx 0.00195$ or $\approx$ 0.2% which is a significant improvement.

Online Micro-clustering

During this process, the incoming information is analyzed and transformed into statistical summaries that are easier to handle. This process is divided in three parts: initialization, classification and assignation. They are described as follows:

The initialization part occurs only once.

Initialization

Before the main maintenance of the micro-clusters, the initialization of q micro-clusters must be performed and each of them is identified with a unique id. The number q is taken as a user input and highly depends on the number of final macro-clusters to be calculated and the capabilities of the system where the algorithm runs. There are different initialization strategies:

- Random initialization:
  1. generate q random d-dimensional vectors to use as centroids of the micro-clusters so that data can be randomly assigned to different micro-clusters.
  2. Assign the first incoming batch of data to their closest random centroids[3].
- KMeans initialization:

[3]See the assignation process for the details.

1. take the first n_i points, number which is specified by the user, and perform the KMeans clustering method over these points for q clusters. By doing this it is possible to ensure that the first n_i points will be better clustered than after a random initialization.
2. Assign the first incoming batch of data to their closest found centroids.

Classification

When a new batch of data arrives, it is necessary to verify point by point whether they belong to a micro-cluster or not. This means that a point has to be located within a factor t of the RMS deviation[4] (RMSD) of the nearest micro-cluster, otherwise it is considered as an outlier.

Outliers are taken into account as potential new micro-clusters; this is because the stream of data might change over time. To do so, one new micro-cluster should be created with an entirely new id, containing the outlier. The number of micro-clusters q is fixed, so one micro-cluster must be freed either by deleting an old micro-cluster of by merging two of them. To decide whether a micro-cluster is safe to be deleted, it is necessary to compute its value of *recency* (RV) and compare it to a user defined threshold δ. If no micro-cluster is safe to delete, then the two closest micro-clusters are merged.

1. Check if a point X_{i_k} fits into a micro-cluster (is not an outlier): $distance(\bar{X}_{i_k}, \hat{M}_j) \leq t \cdot RMSD$, where $\hat{M}_j$ is the nearest micro-cluster.
   - If X_{i_k} is an outlier: compute RV_{M_j}, if $N_j < 2 * m$ then the mean the time stamps is used, otherwise compute an approximation for the last m points of each micro-cluster assuming its points are normally distributed[5] for the $1 - m/2N_j$ percentile.
     a) If $RV_{M_j} < T - \delta$, where RV_{M_j} is the *recency* value for a given micro-cluster and T the current clock time units elapsed since the begin-

[4] The RMSD can be obtained using $\overline{CF2^x}$ and $\overline{CF1^x}$ as shown in 3.3.3

[5] It is possible to derive the mean and the standard deviation from $CF2^t$ and $CF1^t$ as shown in 3.3.3

ning of the stream, then M_j is safe to delete and a new micro-cluster M_{newID} must replace it containing X_{i_k}.

b) When there is no M_j safe to delete, then merge M_j and M_{jj} such that $min\{M_j, M_{jj} : M_j, M_{jj} \in M_Q, distance(M_j, M_{jj})\}$, where M_Q is the set of all micro-clusters. The new merged micro-cluster will now have a list of IDs $\{j, jj\}$ and a new micro-cluster M_{newID} must replace M_{jj} containing X_{i_k}.

2. Assign the point X_{i_k} to $\hat{M}_j$.

Assignation

When a point X_{i_k} is assigned to a micro-cluster, the tuple $(\overline{CF2^x}, \overline{CF1^x}, CF2^t, CF1^t, n)$ for M_j needs to be maintained. Due to the additive property of the micro-clusters it is possible to maintain them directly as follows:

- $\overline{CF2^x}_{new} = \overline{CF2^x}_{old} + \overline{CF2^x}_{X_{i_k}} = \overline{CF2^{x^p}}_{old} + (x^p_{i_k})^2 \; \forall p \in \{1, 2...d\}$
- $\overline{CF1^x}_{new} = \overline{CF1^x}_{old} + \overline{CF1^x}_{X_{i_k}} = \overline{CF2^{x^p}}_{old} + x^p_{i_k} \; \forall p \in \{1, 2...d\}$
- $\overline{CF2^t}_{new} = \overline{CF2^t}_{old} + \overline{CF2^t}_{X_{i_k}} = \overline{CF2^x}_{old} + (T_{i_k})^2$
- $\overline{CF1^t}_{new} = \overline{CF1^t}_{old} + \overline{CF1^t}_{X_{i_k}} = \overline{CF2^x}_{old} + T_{i_k}$
- $n_{new} = n_{old} + 1$

Offline Macro-Clustering

While the online process transforms the stream into statistical summaries in the form of micro clusters, the offline process uses this result to deliver the final macro-clusters[6] for a given time horizon. This means that the micro clusters in the snapshots stored are now the input data for the macro-clustering process.

[6] In K-Means this know as the k clusters.

Assuming that t_c is the current time and h is the user defined horizon, the time window of information would be $(t_c, t_c - h)$. Having snapshots means that probably $t_c - h$ will not exist in an exact form but the snapshot that occurred just before that time is chosen. The pyramidal time frame ensures that it is always possible to find a snapshot $t_c - h'$ within the user specified tolerance for any h.

If $S(t_c - h')$ is the set of micro-clusters at time $t_c - h$ and $S(t_c)$ is the set of micro-clusters at time t_c, it is possible to find the final set of micro-clusters $N(t_c, h')$ by subtracting from $S(t_c)$ each corresponding micro-cluster in $S(t_c - h')$. This is possible due to the fact that each micro-cluster is associated with a list of ids. Doing this ensures that micro-clusters created before the user specified time horizon do not dominate the results of the clustering process.

Property 2.2. *Let C_1 and C_2 be two sets of points. Then the cluster feature vector $\overline{CFT(C_1 \cup C_2)}$ is given by of $\overline{CF2(C_1)} + \overline{CF2(C_2)}$*

Property 2.3. *Let C_1 and C_2 be two sets of points such that $C_1 \supseteq C_2$. Then the cluster feature vector $\overline{CFT(C_1 - C_2)}$ is given by $\overline{CF2(C_1)} - \overline{CF2(C_2)}$*

Properties 2.2 and 2.3 show, respectively, the additive and subtractive nature of the cluster feature vectors. This is particularly helpful as only two snapshots are required to approximate any user specified time horizon or window.

Once $N(t_c, h')$ is constructed, the micro-clusters in it are treated as *pseudo-points*, over which K-Means can be used to determine the higher level and final macro-clusters as follows:

- At its initialization step, the seeds are sampled with probability proportional to the number of points each micro-cluster has instead of picking them randomly, which correspond to the centroids of the micro-clusters.
- The distance from a seed to a *pseudo-point* is equal to the distance between the seed and the centroid of the corresponding micro-cluster.
- When the seeds are adjusted, the new seed is defined as the weighted centroid of the micro-clusters in that partition.

As a matter of fact, following the previous modifications, many traditional clustering algorithms could be used if needed.

2.3. Related work

2.3.1. SAMOA

CluStream has been implemented in different types of software and libraries, being *SAMOA - Scalable Advanced Massive Online Analysis* one of the options. It is also a distributed computing implementation of the algorithm. The difference is that it is not implemented in *Spark*, but rather in a *Distributed Stream Processing Engine* which adapts the *MapReduce* approach to parallel stream processing[17].

Main differences with this adaptation:

- It does not include an offline macro-clustering phase.
- It is developed in *Java* and not designed to work with Spark.

2.3.2. StreamDM

StreamDM is a collection of algorithms for mining big data streams [7]. One of the included methods for stream clustering is *CluStream*. This collection of algorithms is developed for Spark.

Main differences with this adaptation:

- It does not include an offline macro-clustering phase.

[7] As it is stated by them here: http://huawei-noah.github.io/streamDM/

3. Implementation

3.1. Developing environment

The developing environment refers to the collection of software tools required to make this project possible. This section also serves the purpose of giving credit to the people who developed these free-to-use tools by mentioning their work. It is also worth to point out that the selection of this tools was only related to personal preference and does not necessarily mean that they are the best choice for this or other projects.

3.1.1. Operating system

Although a particular operating system (OS) was not indispensable for this project, as the tools used are either open-source or free-to-use and cross-platform, it made the process more comfortable. The OS of choice is *GNU/Linux (or Linux)* and the chosen distribution is *Manjaro KDE*[1].

Manjaro is a distribution of *Linux* based on *Arch*[2], which means it has an on-rolling release program, meaning that it should always be up to date without the need of installing it from scratch whenever a new release comes out. Being based on *Arch* also means that it is possible to use the *Arch User Repository (AUR)*, which is arguably the largest repository for a *Linux* distribution containing almost every piece of software available; this makes the installation of software and setup process of the OS faster an easier.

[1] KDE refers to the desktop environment. More information can be found here: https://manjaro.github.io/
[2] https://www.archlinux.org/

This setup includes:

- Linux kernel 4.1
- Spark 1.5.2
- Scala 2.10.5
- Python 3.5.1
- BASH 4.3

3.1.2. IDE and programming language

IDE stands for integrated developing environment, and in this case *IntelliJ IDEA community edition 15*[3] was the preferred one. This IDE was originally developed for the *Java* programming language but also different versions are available which support other languages.

Spark is being developed using *Scala*[4], a programming language which runs on top of the *Java virtual machine* and brings a couple of useful features like an interactive shell and a closer approach to functional programming, without leaving *Java*'s object oriented programming features out. Therefore, a plugin can be downloaded within the IDE to support *Scala* seamlessly.

3.1.3. Other tools and programming languages

Certain software tools were used less frequently but posses the same importance:

- *Spyder*: an IDE for scientific programming using *Python*. Mainly used for plotting and analyzing data and results.
- *Sublime Text*: a very powerful, general purpose, text editor. It was mostly used for *BASH* scripting.

[3] https://www.jetbrains.com/idea/
[4] http://www.scala-lang.org/

- *Netcat*: tool used to send messages through TCP or UDP connections. Used to stream data to Spark.
- *LibreOffice suit*: spreadsheets an charts were required occasionally.
- *Kile*: the preferred *Latex* editor.

3.2. The global picture

There are two main parts in the CluStream method that need to be adapted: the online microclustering process an the offline macroclustering process. For this work, the online part is the most challenging one as it is the core of the method. Both parts need to be adapted, as they need few modifications in order to properly work in Spark.

Considerations for adapting CluStream to Spark:

Online process: Spark Streaming works with batches of data processed at user-defined time intervals. This means that for each batch all the points are considered to have the same time stamp. Other streaming platforms include time stamps for each arriving data point built in, which would be closer to what the original method[8] refers to. Another reason why every point for each batch has the same time stamp is because once the data arrives, Spark's stream receiver distributes the data points among the cluster for parallel work, and assigning locally a time stamp based on real time might lead to false results for the following reasons:

- Processing units in a cluster can work asynchronously, so that one point that arrived in a more recent time might be processed before an older one.
- Synchronizing the clocks of all processing units can be a challenge on it's own if there is no central source of time.

Offline process: for the offline part it is suggested to use a modified version of K-Means, which involves clustering the micro-clusters with weights depending on the amount of points associated to them. Unfortunately, Spark's K-Means implementation does not openly allow the user to use a weight oriented version of

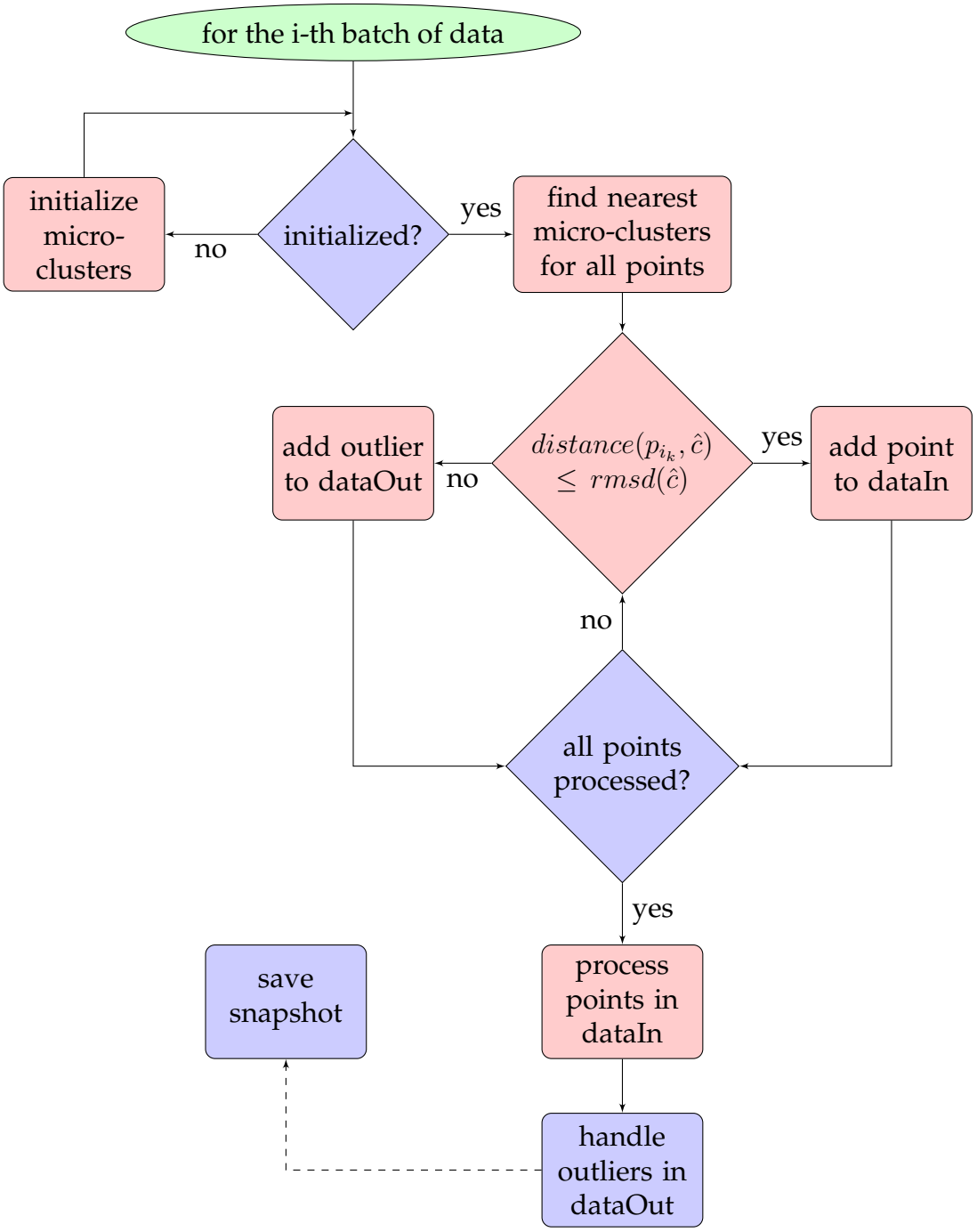


Figure 3.1.: Flow chart of the online micro-clustering process, where p_{i_k} is the k-th point of the i-th batch and $\hat{c}$ is its nearest micro-cluster center

K-Means, and for that reason a new modification is proposed here to circumvent that issue:

- Initialize the centroids by sampling the micro-clusters centers as points using a frequency distribution which can be built based on the number of points in each micro-cluster over the total number points in all of them.
- Instead of using a weighted K-Means version, the previously described distribution can also be used to sample M points and use them as the input of the traditional K-Means.

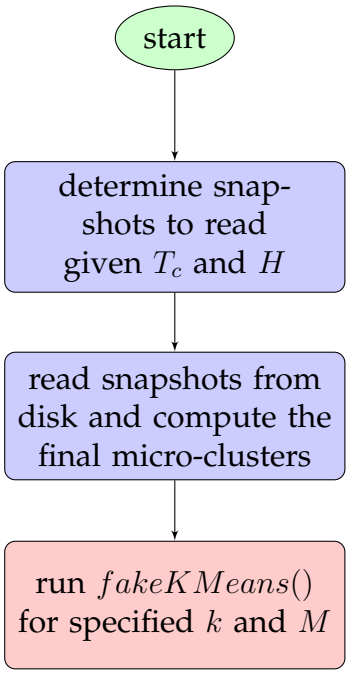


Figure 3.2.: Typical flow chart of the offline macro-clustering process

Figure 3.1 represents a the general idea of the online micro-clustering process. Each box represents a set of actions that are described in section 3.3 of this chapter. Even though this flow chart summarizes this process, it is still possible to observe a greater complexity in comparison to the offline process, represented in Figure 3.2, and that is the reason why more emphasis will be put on the online process

in this chapter. In both Figures, boxes are colored red whenever there are parallel computations within those actions.

3.3. Adapting the CluStream method

In this section, the strategies and details of adapting and implementing the *CluStream* method into Spark are discussed. This task is divided in smaller subtasks taken from the flow charts 3.1 and 3.2 in order to represent more accurately how it was done, i.e. solving one part at a time. Although this was implemented using the *Scala* programming language, not every single line of code are shown in this section, but rather pseudocode and carefully chosen pieces of the code.

3.3.1. Code structure

A total of four classes were designed, two of them hold the main functionality and two are used as helper classes to hold information. The two that hold the functionality are the only ones that the end user would need: the implementations of the online and offline phases are in one separate class each. The classes here described are:

- *CluStreamOnline*: contains the structure to handle DStreams and the necessary functions to process the data and maintain the micro-clusters.
- *CluStream*: contains the functions needed to write, read and process snapshots, as well as the implementation of the modified K-Means version: *fakeK-Means()*.
- *MicroCluster*: contains the elements of the *cluster feature vector* that define a micro-cluster, i.e. $(\overline{CF2^x}, \overline{CF1^x}, CF2^t, CF1^t, n)$.
- *MicroClusterInfo*: contains only the necessary information of the micro-clusters that is needed in the parallel computations[5], such as the centroid, the RMSD

[5]This is discussed in more detail in the performance notes: a smaller object is preferred because it's quicker to broadcast it.

and the size of the micro-clusters.

These classes do not extend each other and therefore the hierarchy of the code is flat, but they do inherit from the *Java* class *Serializable*. Extending this class allows Spark to serialize instances of the objects of these types, this is specially important because it is necessary for sending the objects as messages among all the processing units.

In the original proposed way of using the CluStream method, one separates the offline and online process with the help of saving snapshots. Other implementations, like in *SAMOA*[17], prefer using other strategies which do not involve saving snapshots to disk, and in order to let the user replicate a similar behavior, the implementation regarding snapshots is in the *CluStream* class and it is optional to use. This is the reason why in Figure 3.1 "save snapshots" is directed using a doted line, to remark that it is a function from another class.

Helper classes

To facilitate the understanding of some of the algorithms, first a description of the two helper classes is presented, attributes and functions from these classes are used repeatedly throughout the adaptation. In object oriented programming, it is common to refer to functions as methods and in this section they are equivalent.

Class: MicroCluster	
Constructor: this(cf2x, cf1x, cf2t, cf2t, n, *Array(MicroCluster.inc())*)	
Companion object: var current; inc(){current += 1}	
Attributes:	**Methods:**
var cf2x: breeze.linalg.Vector[Double]	setCf2x(cf2x): Unit, getCf2x(): cf2x
var cf1x: breeze.linalg.Vector[Double]	setCf1x(cf1x): Unit, getCf1x(): cf1x
var cf2t: Long	setCf2t(cf2t): Unit, getCf2t(): cf2t
var cf1t: Long	setCf1t(cf1t): Unit, getCf1t(): cf1t
var n: Long	setN(n): Unit, getN(): n
var ids: Array[Int]	setIds(ids): Unit, getIds(): ids

Figure 3.3.: Abstraction of the MicroCluster class. Code: B.3

As it can be seen in Figure 3.3, the class *MicroCluster* contains all the information that the *cluster feature vector* describes for *CluStream* plus an array of ID's. This array is needed so it is possible to keep track of what happened after two clusters are merged. It also includes a constructor where its inputs are all but the array of IDs (reason why it's on italics in the figure), and the purpose of this constructor is to have an automated way of increasing the ID count every time this constructor is used. This comes very handy because with it there is no need for manually generating unique ID's. The way this was achieved is by declaring a companion object which remains static throughout runtime, multiple instances of these objects are not possible.

The type of most of the attributes are common to many programming languages, the only special type is *breeze.linalg.Vector[Double]*, which is a type of vector within the *breeze* libraries. These libraries allow fast linear algebra operations which can use an implementation of *BLAS* or *LAPACK* if it is properly configured in the system, otherwise it falls to an implementation in *Java* called *F2jBlAS*. Diverse functions from these libraries are used in this adaptation.

Class: MicroClusterInfo	
Attributes:	**Methods:**
var centroid: breeze.linalg.Vector[Double]	setCentroid(centroid): Unit
var rmsd: Double	setRmsd(rmsd): Unit
var n: Long	setN(n): Unit

Figure 3.4.: Abstraction of the MicroClusterInfo class. Code: B.4

The *MicroClusterInfo* class, Figure 3.4, is much simpler, it only needs to hold the necessary information that will be sent to all nodes in the cluster to perform parallel computations. For example, the centroid is required to determine the nearest micro-cluster to a given point and the RMSD to determine whether this point is an outlier or belongs to the micro-cluster in question.

3.3.2. Level of parallelism

The strategy to parallelize code often requires compromises, for example: whether communication costs are worth less than computation time or not. In this case the decision was to maintain the micro-clusters locally but perform as many computations in parallel as possible with the intention to reduce communication costs and to avoid broadcasting up-to-date information for every point processed as much as possible. This comes with some consequences:

- An extra array containing critical information has to be created and maintained, following the logic that sending the whole collection of micro-clusters is communication-wise costly.
- Some operations must be implemented in serial code, such as the maintenance of the array mentioned in the point above and the handling of the outliers because it requires to modify the current micro-clusters in every iteration (for each outlier).
- Performance will be highly dependent on the amount of outliers for a given batch. For example, if the data is noisy or rapidly changing there will be more outliers.

For the offline phase, storing snapshots and reading from them is also done serially due to the fact that the micro-clusters are located in local memory. The $fakeKMeans()$ algorithm runs on parallel as it uses Spark's implementation of K-Means, but having to read the snapshots locally means that the data has to be distributed before it runs.

3.3.3. CluStreamOnline class (online phase)

In this section, the five main components of the online process are described. Naturally, this is a generic overview of how each component is implemented, the complete code is in the Appendix B.1. These parts are abstracted from the flow chart in Figure 3.1, which are: initialization, finding nearest micro-cluster, finding outliers, processing points and handling outliers.

Initialization

The name of variables in these sections are in italics.

For this adaptation, there are three initialization options. The first one is to generate random q micro-clusters centroids, the second one is to take the first *minInitPoints* to run a standard K-Means to cluster those micro-clusters and the third one is to use Spark's Streaming K-Kmeans to update the centroids until at least $minInitPoints$ have been clustered. All initialization options, and in fact the main loop, follow similar steps. The Streaming K-Means initialization is not demonstrated in this section due to similarities. The general algorithm to randomly initialize the micro-clusters is detailed in Algorithm 3.

Algorithm 3 Random initialization. Code: B.1.1

Input: *rdd: RDD[breeze.linalg.Vector[Double]]*— this RDD comes directly from the DStream in the main loop.
Abbreviations: *McI = new MicroClusterInfo*

1: $mcInfo \leftarrow \{(McI(rand()), 0)_1, ..., (McI(rand()), q-1)_q\}$
2: $assignations \leftarrow assignToMicroCluster(rdd, mcInfo)$
3: $updateMicroClusters(assignations)$
4: $updateMicroClusterInfo()$
5: $broadcastMCInfo \leftarrow rdd.context.broadcast(mcInfo)$
6: $initialized \leftarrow true$

Some important remarks:

- *mcInfo* is collection of q (number of micro-clusters) tuples, and each tuple contains a *MicroClusterInfo* instance and an integer which corresponds to a unique ID to identify its respective micro-cluster (or *MicroCluster*). This is used later on to find nearest micro-clusters and outliers.
- *McI(rand())* only means that the centroid is constructed randomly.
- Lines 2 and 3 in Algorithm 3, represent the main procedures to maintain the micro-clusters. These instructions are repeated again in Algorithm 4 and are, in fact, the core of the main loop for every batch of data processed[6].

[6]In the diagram of Figure 3.1 this would be from "find nearest micro-clusters for all points" to "handle outliers"

- Lines 4 and 5 from the same Algorithm, represent the instructions executed to update the *mcInfo* and broadcast it.

Algorithm 4 Initialization with K-Means. Code: B.1.2

Input: *rdd: RDD[breeze.linalg.Vector[Double]]*— this RDD comes directly from the DStream in the main loop.
Abbreviations: *McI = new MicroClusterInfo*

1: $initArray \leftarrow initArray.append(rdd.collect)$
2: **if** $initArray.lenght \geq minInitPoints$ **then**
3: $trainingSet \leftarrow rdd.parallelize(initArray)$
4: $clusters \leftarrow KMeans.train(trainingSet, q)$
5: $mcInfo \leftarrow \{(McI(clusters_0), 0)_1, ..., (McI(clusters_{q-1}), q-1)_q\}$
6: $assignations \leftarrow assignToMicroCluster(rdd, mcInfo)$
7: $updateMicroClusters(assignations)$
8: $updateMicroClusterInfo()$
9: $broadcastMCInfo \leftarrow rdd.context.broadcast(mcInfo)$
10: $initialized \leftarrow true$
11: **end if**

Algorithm 4, which is the K-Means initialization, differs from the other one as it is called for every batch as long as the *initialized* variable remains false and it will only initialize the micro-clusters when there are enough points to perform the K-Means clustering algorithm, and this Spark implementation is part of Spark MLlib and it's designed to run in parallel. These points are temporarily stored in $initArray$ until the condition of having at least $minInitPoints$ is met, this is done via the RDD action $collect$, which only means to get all the distributed elements of the RDD and bring them as an array to the driver node[7]. Centroids are taken then from the result of this clustering process and are used to initialize the *mcInfo* variable.

There are some functions in both algorithms that are common and, as mentioned, used also in the main loop for maintaining the micro-clusters. These are detailed in the following sections but in summary:

[7]The driver node is the one running the Spark application that eventually collects results

1. $assignToMicroCluster(rdd, mcInfo)$ has the purpose of finding the nearest micro-cluster to each point and returns a tuple containing the point itself and a unique ID to identify the nearest micro-cluster.

2. $updateMicroClusters(assignations)$ has the purpose of separating the assignations done in the previous step into outliers and points satisfying the condition described in 2.2.2. It also processes the points and handles the outliers.

3. $updateMicroClusterInfo()$ takes the updated *MicroCluster* instances and calculates their centroids and RMSDs of them in order to broadcast up-to-date information. This is described in the Appendix A.2.

Finally, *rdd.context.parallelize()* and *rdd.context.broadcast()* are functions of a Spark context, which is an object containing information about the Spark environment, such as the location of *Master node* of the cluster, the name of the application, number of cores to use, etc. As RDDs are parallelized arrays, they belong to a specific Spark context. The *parallelize()* function distributes an array through the cluster in different partitions, while *broadcast()* distributes an object equally to all nodes in the cluster.

Finding nearest micro-cluster

The maintenance of the micro-clusters starts with this operation. Once initialization is performed, finding the nearest micro-clusters for all the points is the very first thing to be done for every new batch of data. Specifically, the function in charge of doing that is $assignToMicroCluster()$, discussed superficially before.

Finding the nearest micro-clusters is an operation of complexity $O(n * q * d)$, where n is the number of points, q the number of micro-clusters and d the dimension of the points; q and d remain constant during runtime but n might vary. Algorithm 5 describes a simple search for the minimum distance for every point in the RDD to the micro-clusters. This is also a good opportunity to show how this works using Spark and *Scala*:

Algorithm 5 Find nearest micro-cluster ($assignToMicroCluster()$). Code: B.1.20

Input: *rdd: RDD[breeze.linalg.Vector[Double]], mcInfo: Array[(McI,Int)]*— *rdd* is an RDD containing data points and *mcInfo* is the collection of the micro-clusters information.

Output: *rdd: RDD[(Int, breeze.linalg.Vector[Double])]* — returns a tuple of the point itself and the unique ID of the nearest micro-cluster.

```
1:  for all p ∈ rdd do
2:      minDistance ← Double.PositiveInfinity
3:      minIndex ← Int.MaxValue
4:      for all mc ∈ mcInfo do
5:          distance ← squaredDistance(p, mc_1.centroid)
6:          if distance ≤ minDistance then
7:              minDistance ← distance
8:              minIndex ← mc_2
9:          end if
10:     end for
11:     p = (minIndex, p)
12: end for
        return rdd
```

```
1  def assignToMicroCluster(rdd: RDD[Vector[Double]], mcInfo: Array[(
       MicroClusterInfo, Int)]): RDD[(Int, Vector[Double])] = {
2    rdd.map { a =>
3      var minDist = Double.PositiveInfinity
4      var minIndex = Int.MaxValue
5      for (mc <- mcInfo) {
6        val dist = squaredDistance(a, mc._1.centroid)
7        if (dist < minDist) {
8          minDist = dist
9          minIndex = mc._2
10       }}
11     (minIndex, a)
12   }}
```

Scala allows operating the elements of a collection, e.g. an array, through a map operation. The code above represents the actual function in the source code of this project that does what Algorithm 5 says for the initialization process, the only difference with this implementation from the one in the main loop is that $mcInfo$ does not get passed because it has been already broadcasted. In this programming language, the last line defines what the function returns; it can be seen in line 2 of the mentioned function that there is actually only one instruction called, and that is $rdd.map\{a => function(a)\}$. This operation, *map*, requires a function to be passed, which at the same time receives as argument each element a of a collection, in this case the collection is rdd, and returns anything resulting from that function which will replace a. In other words, it is possible to operate and change every element of a collection through map with a specified function and get a new "updated" collection in return.

To find the nearest micro-cluster of a point, in this case a, an iterative process is used, which computes the squared distance from a to all micro-clusters' cetroids and stores only smaller distances and the unique ID of such so-far-nearest micro-cluster. When the iteration finishes, the function returns the tuple $(minIndex, a)$ which replaces the element a.

Spark uses this map operation to serialize the function passed so that all nodes

in the cluster get the same instruction, this is exactly how computations are parallelized within this framework. At this point, every node performs this operation to find the nearest micro-cluster for all the points they locally have.

Finding outliers

Once every point has been assigned to its nearest micro-cluster, it is necessary to find all those points which do not lie within the specified factor of the RMSD of such micro-cluster. The *CluStream* method indicates to compute the RMSD whenever a micro-cluster has more than 1 point in it; however, it does not indicate how this can be done, fortunately the information contained in the *cluster feature vector* is sufficient, as it is shown in the following proof:

Proof: RMSD from cluster feature vector.

The RMSD is defined as the square root of the mean squared error,

$$RMSD = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - c)^2}, \tag{3.1}$$

expanding 3.1:

$$RMSD = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i^2 - 2x_i c + c^2)} = \sqrt{\frac{1}{N}\left(\sum_{i=1}^{N}x_i^2 - 2c\sum_{i=1}^{N}x_i + c^2\sum_{i=1}^{N}(1)\right)}, \tag{3.2}$$

where x_i and c are vectors of a point and its nearest centroid, respectively, and N the number of points of the micro-cluster. From the *cluster feature vector* definition:

$$\overline{CF1^x} = \left[\sum_{i=1}^{N}x_i^1, ..., \sum_{i=1}^{N}x_i^d\right]^T, \tag{3.3}$$

$$\overline{CF2^x} = \left[\sum_{i=1}^{N}(x_i^1)^2, ..., \sum_{i=1}^{N}(x_i^d)^2\right]^T, \tag{3.4}$$

from 3.3, it is possible to get the centroid:

$$c = \frac{1}{N}\overline{CF1^x}, \tag{3.5}$$

as x_i is a vector, and for a vector v, v^2 is equal to $\|v^2\|_2^2 = v \cdot v$, therefore:

$$\sum_{i=1}^{N} x_i^2 = \sum_{i=1}^{N} x_i \cdot x_i = \|\overline{CF2^x}\|_1, \tag{3.6}$$

being $\|\cdot\|_1$ and $\|\cdot\|_2$ the L^1 and L^2 norms respectively. And again, because x_i is a vector:

$$\sum_{i=1}^{N} x_i = \overline{CF1^x}, \tag{3.7}$$

substituting 3.5, 3.6, 3.7 in 3.2:

$$\begin{aligned} RMSD &= \sqrt{\frac{1}{N}\left(\|\overline{CF2^x}\|_1 - \frac{2}{N}\overline{CF1^x}\cdot\overline{CF1^x} + \left(\frac{1}{N}\overline{CF1^x}\right)\cdot\left(\frac{1}{N}\overline{CF1^x}\right)N\right)} \\ &= \sqrt{\frac{1}{N}\left(\|\overline{CF2^x}\|_1 - \frac{2}{N}\|\overline{CF1^x}\|_2^2 + \frac{1}{N}\|\overline{CF1^x}\|_2^2\right)}, \end{aligned} \tag{3.8}$$

and finally:

$$RMSD = \sqrt{\frac{1}{N}\|\overline{CF2^x}\|_1 - \frac{1}{N^2}\|\overline{CF1^x}\|_2^2}, \tag{3.9}$$

which proves that the RMSD can be obtained using only the *cluster feature vector*. □

The process of finding outliers happens at the beginning of the *updateMicroClusters()* function, which takes as argument the resulting RDD from *assignToMicroCluster()* which contains the assignations of the points to their nearest microcluster.

This operation is performed in parallel, taking advantage that the assignations have been done and they are still distributed. The variable $dataInAndOut$ is the RDD resulting from performing a map on the $assignations$ RDD, which is the input

Algorithm 6 Find outliers ($updateMicroClusters()$ pt. 1). Code: B.1.22

Input: *assignations: RDD[breeze.linalg.Vector[Double]]*— *assignations* is an RDD resulting from $assignToMicroCluster()$.

1: $dataInAndOut \leftarrow \{$
2: **for all** $a \in assignations$ **do**
3: $\quad nearMCInfo \leftarrow broadcastMCInfo.value.find(id => id_2 == a_1)$
4: $\quad nearDistance \leftarrow sqrt(squaredDistance(a_2, nearMCInfo.centroid))$
5: $\quad$ **if** $nearDistance \leq t * nearMCInfo.rmsd$ **then**
6: $\quad\quad$ **return** $(1, a)$
7: $\quad$ **else**
8: $\quad\quad$ **return** $(0, a)$
9: $\quad$ **end if**
10: **end for**
11: $\}$
12: $dataIn \leftarrow dataInAndOut.filter(val_1 == 1).get(val_2)$
13: $dataOut \leftarrow dataInAndOut.filter(val_1 == 0).get(val_2)$

of Algorithm 6. The RMSD value is needed to check whether a point lies inside or outside a micro-cluster, therefore the broadcasted object $broadcastMCInfo$ is used to get this information. Then it is only a matter of comparing the distance of a point to its nearest micro-cluster to the RMSD value times the factor t, which is defined by the user. If the point lies inside, then it returns the tuple $(1, a)$ and if not $(0, a)$, where a is an element of the $assignations$ RDD.

At the end of this algorithm, two new RDDs are defined $dataIn$ and $dataOut$, for which the previous result gets filtered as shown in lines 12 and 13: $dataIn$ gets the second elements of the tuples of $dataInAndOut$ where the first element is 1 and, similarly, $dataOut$ gets the ones for which the first element of the tuple is 0.

Processing points

After the points have been separated, it is possible to compute the necessary information from them to update the micro-clusters. It is important to perform this step before handling the outliers because this adaptation process the points for batches of data and not points individually as they arrive, and the reasons are:

- Every point for a batch is treated equally in terms of temporal properties. A batch of points gets distributed among the cluster and they all get the same time stamp. As the points are processed in parallel, and there is no constant communication among nodes, there is no reason to assume that a point is older or newer within that batch because then race conditions would occur resulting in unpredictable results.
- Taking into account the previous statement, the process of handling outliers involve deleting and merging micro-clusters and not processing the points first would lead to invalid assignations as some micro-clusters might not exist anymore or might have changed while being merged.

These two points are some of the key differences between the original $CluStream$ method and this adaptation. For the original, it is possible to handle point by point as each have different clear time stamps.

Processing the points means two things: to compute the values required to update the micro-clusters, i.e. the *cluster feature vector*, and actually updating the micro-clusters. This is another good opportunity to show a piece of code because it will be important for the performance analysis later on, and also to show how it is possible to reduce the amount of code needed for such operation. First, computing the *cluster feature vector*:

```
1 val aggregateFunction = (aa: (Vector[Double], Vector[Double], Long),
2                          bb: (Vector[Double], Vector[Double], Long)) =>
3         (aa._1 :+ bb._1, aa._2 :+ bb._2, aa._3 + bb._3)
4
5 val sumsAndSumsSquares = dataIn.mapValues(v => (v, v :* v, 1L))
6         .reduceByKey(aggregateFunction).collect()
```

First a function called $aggregateFunction$ is defined, and this is a generic function that can get passed as an argument to another function. It has the purpose of taking any two given values, in this case two given tuples, and return a tuple that contains the sum of every element of one tuple with the respective element of the other tuple:

$$aggregateFunction = ((v_1, u_1, k_1), (v_2, u_2, k_2)) => (v_1 + v_2, u_1 + u_2, k_1 + k_2)$$

From the assignation process, the points that are going to be processed are located in $dataIn$, which looks as follows:

$$dataIn = \{(id_1, p_1), (id_2, p_2), ...\},$$

where, id_i is the unique identifier of the micro-cluster the point p_i belongs to. Then a map is performed only on the "values" of the tuple, considering that Spark can interpret tuples as $(key, value)$ pairs, to replace each point p_i with a tuple containing p_i, squared elements of p_i and the *Long* value of 1:

$$dataIn.mapValues(v => (v, v * v, 1)) = \{(id_1, (p_1, p_1^T I p_1, 1)), (id_2, (p_2, p_2^T I p_2, 1)), ...\}$$

To square the elements means to element-wise square the values of a vector v, therefore this is represented by $v^T I v$. This is done in order to perform a $reduceByKey$ operation, which is how Spark combines and operates the distributed elements of an RDD:

$$dataIn.mapValues(...).reduceByKey(aggregateFunction) =$$
$$\{(id_1, (p_{1,1} + p_{1,2} + ..., p_{1,1}^T I p_{1,1} + p_{1,2}^T I p_{1,2} + ..., 1 + 1 + ...)), ...\}$$

The $(key, value)$ pairs are important here because then all the points belonging to the same micro-cluster are "reduced" together, resulting in tuples containing the element-wise sum, square sum and count of points: $(id, (\overline{CF1^x}_{new}, \overline{CF2^x}_{new}, N_{new}))$. After having computed the values to update the *cluster feature vector* of the micro-clusters which get new points, it is possible to proceed to do so as in Algorithm 7.

Algorithm 7 Update micro-clusters ($updateMicroClusters()$ pt. 2). Code: B.1.22

```
1: for all mc ∈ microClusters do
2:     for all ss ∈ sumsAndSumsSquares do
3:         if mc.getIds(0) == ss_1 then
4:             mc.setCf1x(mc.cf1x : +ss_{2_1})
5:             mc.setCf2x(mc.cf2x : +ss_{2_2})
6:             mc.setN(mc.n + ss_{2_3})
7:             mc.setCf1t(mc.cf1t + ss_{2_3} * time)
8:             mc.setCf2t(mc.cf2t + ss_{2_3} * (time * time))
9:         end if
10:    end for
11: end for
```

Handling outliers

Whenever there are outliers in a batch of data, these points need to be treated as potential new clusters because the stream might change over time. As described in 2.2.2, after a micro-cluster has enough points ($N > 2 * m$), where N is the number of points in a particular micro-cluster and m is a user defined parameter, the mean time stamp is no longer used as the value of recency and an approximation needs to be computed. To accomplish this, the mean and the standard deviation of the time stamps are required to get this approximated value of recency, assuming the time stamps are normally distributed. While the mean is rather straight forward to compute, the standard deviation is a little bit more involved. Fortunately the sum and the sum of squares of the time stamps are stored for each micro-cluster and it is possible to obtain those values:

Proof: standard deviation from $CF1^t$ and $CF2^t$.

The standard deviation is defined as:

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(t_i - \mu)^2}, \tag{3.10}$$

expanding 3.10:

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(t_i^2 - 2t_i\mu + \mu^2)} = \sqrt{\frac{1}{N}\left(\sum_{i=1}^{N}t_i^2 - 2\mu\sum_{i=1}^{N}t_i + \mu^2\sum_{i=1}^{N}(1)\right)}, \tag{3.11}$$

where t_i and μ are scalars of a time stamp and the mean time stamp, respectively, and N the number of points of the micro-cluster. From the $CF1^t$ and $CF2^t$ definitions:

$$CF1^t = \sum_{i=1}^{N} t_i, \tag{3.12}$$

$$CF2^t = \sum_{i=1}^{N} t_i^2, \tag{3.13}$$

from 3.12, it is possible to get the mean time stamp:

$$\mu = \frac{1}{N}CF1^t, \tag{3.14}$$

substituting 3.14, 3.13, 3.12 in 3.11:

$$\begin{aligned}\sigma &= \sqrt{\frac{1}{N}CF2^t - \frac{2}{N}CF1^t\mu + \frac{N}{N}\mu^2} \\ &= \sqrt{\frac{1}{N}CF2^t - 2\mu^2 + \mu^2},\end{aligned} \tag{3.15}$$

and finally:

$$\sigma = \sqrt{\frac{1}{N}CF2^t - \mu^2} = \sqrt{\frac{1}{N}CF2^t - \left(\frac{CF1^t}{N}\right)^2}, \tag{3.16}$$

which proves that the standard deviation can be obtained using only $CF1^t$ and $CF2^t$. □

In an ideal scenario, the last m points of each micro-cluster would be stored in order to always get an exact recency value with their mean, but that implies increasing the storage requirements. Instead of storing these points, an approximation of that is computed: the idea is to compute the time stamp for the $1-m/2N$

percentile. In Figure 3.5, a representation of normally distributed time stamps is shown along with its normal distribution, it can be seen that the approximation is fairly accurate if the time stamps are actually normally distributed.

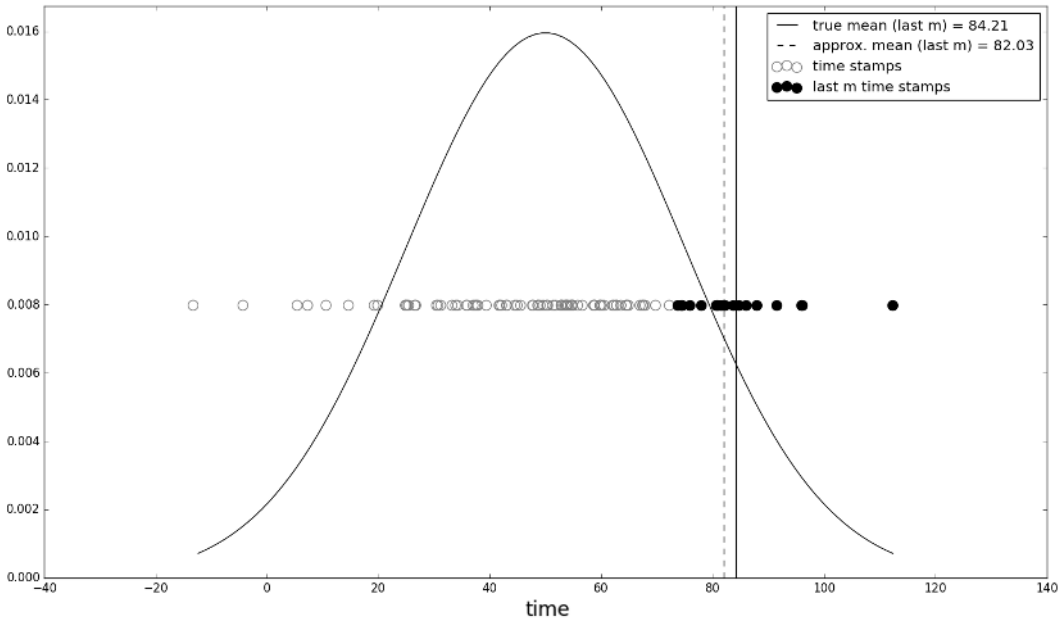


Figure 3.5.: The approximated recency value for the $1 - m/2N$ percentile

To compute the actual approximation of the mean time stamp for the last m points for a given percentile, the *Quantile function*, or *inverse cumulative distribution function*, of the normal distribution is used. The *breeze* libraries also have a statistics set of functions that are helpful in this case, because it is possible to compute this value by only providing the mean, the standard deviation and the percentile as shown in line 8 of Algorithm 8.

First, two arrays are created, one to store the local indexes of the array containing all the micro-clusters $microClusters$, which also matches the $mcInfo$ array, that are safe to delete and the other one for the rest. The local indexes are used in this procedure because handling the outliers is performed locally an not in parallel, so there is no need to look for the unique ID of the micro-clusters. In Algorithm 8, the $mTimeStamp$, which is the recency value, is compared to the recency threshold

Algorithm 8 Find safe to delete micro-clusters ($updateMicroClusters()$ pt. 3). Code: B.1.22

1: $i \leftarrow 0$
2: **for all** $mc \in microClusters$ **do**
3: $mean \leftarrow mc.getCf1t/ms.getN$
4: $stdDev \leftarrow sqrt(mc.getCf2t/mc.getN - mean * mean)$
5: **if** $mc.getN < 2 * m$ **then**
6: $mTimeStamp \leftarrow mean$
7: **else**
8: $mTimeStamp \leftarrow Gaussian(mean, stdDev).icdf(1 - m/2 * mc.getN))$
9: **end if**
10: **if** $mTimeStamp < time - delta$ **then**
11: $safeDelete.append(i)$
12: **else**
13: $keepOrMerge.append(i)$
14: **end if**
15: $i \leftarrow i + 1$
16: **end for**

$time - delta$ to determine whether the micro-cluster in question is old enough to be safe to delete or not, where $time$ is the current time unit and $delta$ is a user defined parameter.

Finally, knowing which micro-clusters are safe to delete, the outliers can be handled. The first thing that happens in this operation, is to check whether an outlier can be absorbed by a newly created micro-cluster as a result from other outlier, this compensates an issue which batch processing brings: if this does not happen, then equal (or extremely near) points would create a new micro-cluster of their own, not resembling the behavior of the original method. The first outlier skips this step simply because the array $newMicroClusters$ is empty, and only grows every time a new micro-cluster is created. In general, there are three possible scenarios:

- If the point lies within the restriction regarding the RMSD for its nearest micro-cluster in $newMicroClusters$, the point is added to it.
- If the point does not lie within any of the new micro-clusters, then it re-

Algorithm 9 handle outliers ($updateMicroClusters()$ pt. 4). Code: B.1.22

```
1: j ← 0
2: for all p ∈ dataOut do
3:     distance, mcID ← getMinDistanceFromIDs(newMicroClusters, p_2)
4:     if distance < t * mcInfo[mcID]_1.rmsd then
5:         addPointToMicroCluster(mcID, p_2)
6:     else
7:         if safeDelete[j].isDefined then
8:             replaceMicroCluster(safeDelete[j], p_2)
9:             newMicroClusters.append(j)
10:            j ← j + 1
11:        else
12:            index1, index2 ← getTwoClosestMicroClusters(keepOrMerge)
13:            mergeMicroClusters(index1, index2)
14:            replaceMicroClusters(index2, p_2)
15:            newMicroClusters.append(j)
16:            j ← j + 1
17:        end if
18:    end if
19: end for
```

places a micro-cluster from the $safeDelete$ array, assuming there are safe-to-delete micro-clusters. This is done until every safe-to-delete micro-cluster is deleted. There is no further method to prioritize deletions.

- If none of the previous scenarios are viable, then the two mirco-clusters that are closest to each other get merged, freeing one spot to create the new micro-cluster. This is the the most computationally expensive scenario. The function $getTwoClosestMicroClusters()$ has a complexity of $O(p_m d \cdot \frac{q!}{2!(q-2)!})$, where p_m is the number of outliers that require a merge, d the dimension of the points, and q the number of micro-clusters.

It is important in the procedure described in Algorithm 9 to locally update the $mcInfo$ every time a point is added to a micro-cluster, two micro clusters are merged and when a new micro-cluster is created. There could be a lot of change,

depending on the outliers, and this loop requires up-to-date information for each iteration, otherwise merges and the RMSD check would be inaccurate.

Main loop, putting it all together

The procedure in Algorithm 10 is executed every time a batch of data is processed, that is the main loop. The input is the DStream that is generated by the Spark streaming receiver. The only to conditions which need to be satisfied to execute these instructions are that the batch of data, here an RDD, must contain points, i.e. $rdd.count() > 0$, and that the initialization process is completed. Regardless of the batch containing points, the local time is always increased.

Algorithm 10 Main loop (for every batch of data). Code: B.1.4

Input: *data: DStream[breeze.linalg.Vector[Double]]*— this is the data coming directly from the stream receiver

for all $rdd \in data$ **do**
  if $rdd.count() > 0$ **then**
    if $initialized$ **then**
      $assignations \leftarrow assignToMicroCluster(rdd, mcInfo)$
      $updateMicroClusters(assignations)$
      $updateMicroClusterInfo()$
      $broadcastMCInfo \leftarrow rdd.context.broadcast(mcInfo)$
    else
      $initialize()$
    end if
  end if
  $time = time + 1$
end for

3.3.4. CluStream class (offline phase)

In this section, the final clustering process is described. This strategy allows the user to separate the stream micro-clustering from the macro-clustering phase if

desired. The idea is to save snapshots of the current micro-clusters at particular times, following a pyramidal pattern as described in 2.2.2.

Saving snapshots

In order to determine whether a snapshot for the current time $time$ is saved on disk or not, it is necessary to compute if the current time fulfills the conditions defined by the pyramidal pattern and at the same time determine whether a snapshot needs to be deleted: this is due to the fact that only $\alpha^l + 1$ snapshots are stored for each order i, where α, l are user defined parameters.

Algorithm 11 Saving snapshots ($saveSnapshots()$). Code: B.2.2

Input: *dir: String, tc: Long, alpha: Int, l: Int*— *dir* is the directory where the snapshots are saved, tc is the current time and $alpha, l$ are user defined parameters.

1: $write \leftarrow false$
2: $mcs \leftarrow model.getMicroClusters$
3: $limit \leftarrow log(tc)/log(alpha)$
4: **for all** $i \in \{0, 1, ..., limit\}$ **do**
5: **if** tc mod $alpha^{i+1} != 0$ AND tc mod $alpha^i == 0$ **then**
6: $order \leftarrow i$
7: $write \leftarrow true$
8: **end if**
9: **end for**
10: **if** $write$ **then**
11: $File(dir + "/" + tc).wirteObject(mcs)$
12: **end if**
13: $tcDelete \leftarrow tc - (alpha^l + 1) \cdot alpha^{order+1}$
14: **if** $tcDelete > 0$ **then**
15: $File(dir + "/" + tcDelete).detele()$
16: **end if**

Algorithm 11 shows how to determine when to save a snapshot and when to delete it. It is assumed that this function is executed after every batch of data is processed. More details about this algorithm:

- $model.getMicroClusters$ is a function that returns the current array of micro-clusters, where $model$ is the object enclosing the online streaming process.
- In line 3, the limit of the loop that searches the correct order a snapshot would belong to is calculated. This is important to avoid extra computations. The operation $log(tc)/log(alpha)$ is equivalent to $log_{alpha}(tc)$.
- The condition in line 5 ensures to search only for non-redundant orders, fulfilling the conditions established in 2.2.2. The fact that the loop goes in increasing order for i also ensures to always get the maximum possible order for the given tc.
- It was decided to store the micro-clusters as objects to avoid converting them to text and parsing the text to redefine the micro-clusters while reading the files. Also, these object are written in binary *Scala* code with their respective object headers: this simplifies the process of reading the files.
- Line 13 gets the "time" it needs to be deleted to only keep $alpha^l+1$ snapshots for any order i. If $tcDelete$ is negative, the order i is still not full.

Reading the snapshots

The idea behind reading the snapshots goes further than literally only reading files. To get the final clusters, it is possible to define a time window to do so. clustering a time window means that only what happened in that specific time window is relevant, therefore it is desired to eliminate anything that happened after and before that time window.

Figure 3.6 demonstrates the idea of the time window: for a horizon $H = 5$ and a current time of $tc = 100$, ideally the only information that should be clustered is the one between what happened in time 100 and 95. Thanks to Property 2.3 of the *cluster feature vector*, it is possible to achieve this requiring only two snapshots. A given snapshot S_{tc} represents the entire evolution of the stream from time 0 to tc, therefore if the information contained in a snapshot $S_{tc-H} = S_{100-5} = S_{95}$ is subtracted to the information of the more recent snapshot $S_{tc} = S_{100}$, the result

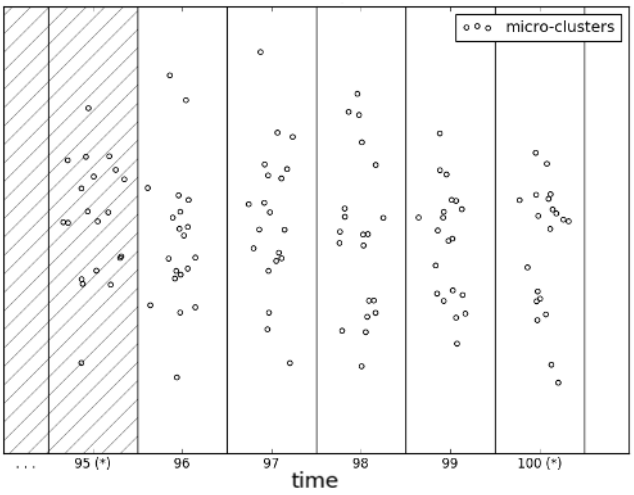


Figure 3.6.: Time window for $tc = 100$ and $H = 5$

would be exactly the information of what happened between $tc - H$ and tc. In the figure, the grayed area is the information to be eliminated.

To be able to get an array of micro-clusters for a give time window, first the real names of the files containing the snapshots need to be obtained, that is the purpose of $getSnapshots()$. Then, after reading the files, the IDs of the micro-clusters are used to determine whether a more recent micro-cluster existed at time $tc - H$, and sometimes more than one micro-cluster in the past is related to a more recent one due to merges: $findRelatedMcsFromIDs()$ represents this procedure. Finally, for every micro-cluster in tc, a subtraction of all its properties is performed for each micro-cluster in $tc - H$ that had an influence on it, i.e. whenever they share IDs, resulting in a new array of micro-clusters containing the exact information for the given time window.

The fakeKMeans solution

The original *CluStream* method suggests to use a slightly modified version of K-Means, a version for which one can initialize the seeds (initial clusters) by sampling from the micro-clusters' centroids taking into account the number of points each

Algorithm 12 Reading snapshots ($getMCsFromSnapshots()$). Code: B.2.4

Input: *dir: String, tc: Long, h: Int*— *dir* is the directory where the snapshots are saved, tc is the current time and h is the horizon.

Output: $mcs : Array[MicroCluster]$ — returns the array of micro-clusters for the specified time window.

1: $(t1, t2) \leftarrow getSnapshots(dir, tc, h)$
2: $snap1 \leftarrow File(dir + "/" + t1).read()$
3: $snap2 \leftarrow File(dir + "/" + t2).read()$
4: $relatingMCs \leftarrow findRelatedMcsFromIDs(snap1, snap2)$
5: **for all** $mc \in relatingMCs$ **do**
6: **for all** $id \in mc_2$ **do**
7: $mc_1.setCf2x(mc_1.getCf2x : -snap2(id).getCf2x)$
8: $mc_1.setCf1x(mc_1.getCf1x : -snap2(id).getCf1x)$
9: $mc_1.setCf2t(mc_1.getCf2t - snap2(id).getCf2t)$
10: $mc_1.setCf1t(mc_1.getCf1t - snap2(id).getCf1t)$
11: $mc_1.setN(mc_1.getN - snap2(id).getN)$
12: $mc_1.setIds(mc_1.getIDs.remove(snap2(id).getIds))$
13: **end for**
14: **end for**
return $relatingMCs$

micro-cluster has and for which one can use these centroids as weighted input points. These weights, again, are related to the number of points they absorbed. Spark's (current) implementation of K-Means does allow to initialize the seeds but unfortunately it is not possible to predefine the weights for the input points.

Algorithm 13 The fakeKMeans algorithm ($fakeKMeans()$). Code: B.2.7

Input: *sc: SparkContext, k: Int, n: Int, mcs:Array[MicroCluster]*— *sc* is the Spark Context in which the clustering is performed, k is the number of macro-clusters, n is the number of points to be sampled and mcs is the array of micro-clusters.

Output: $model : KMeansModel$ — returns the K-Means model resulting from the clustering process. This model is default to Spark.

1: $centers \leftarrow getCentersFromMC(mcs)$
2: $weights \leftarrow getWeightsFromMC(mcs))$
3: $points \leftarrow \{sample(centers, weights)_1, ..., sample(centers, weights)_n\}$
4: $initialClusters \leftarrow \{sample(centers, weights)_1, ..., sample(centers, weights)_k\}$
5: $KMeans.setK(k)$
6: $KMeans.setInitialModel(initialClusters)$
7: $trainingSet \leftarrow sc.parallelize(points)$
8: $model \leftarrow KMeans.run(trainingSet)$
return $model$

In order to solve this issue, a new version of K-Means needs to be implemented. This version uses, in fact, Spark's own version, but to overcome the problem of not being able to define the weights at the beginning, this new version uses as input many points sampled from the micro-clusters' centroids. Algorithm 13 shows this procedure. Remarks on the $fakeKMeans$ algorithm:

- The $getCentersFromMC()$ function returns an array with the centroids of the micro-clusters as follows: for each micro-cluser the operation $\frac{1}{N}\overline{CF1^x}$ is performed, where N is the number of points of the micro-cluster in question.

- The $getWeightsFromMC()$ function returns an array with the weights of the micro-clusters as follows: for each micro-cluser the operation $\frac{N}{totalPoints}$ is performed, where N is the number of points of the micro-cluster in question and

$totalPoints$ is the sum of all N's. By doing this, a frequency distribution is generated.

- The $sample()$ function takes the centroids and their weights to randomly sample centroids for the given frequency distribution: the more points a micro-cluster has, the more likely its centroid will be sampled, as shown in Figure 3.7.

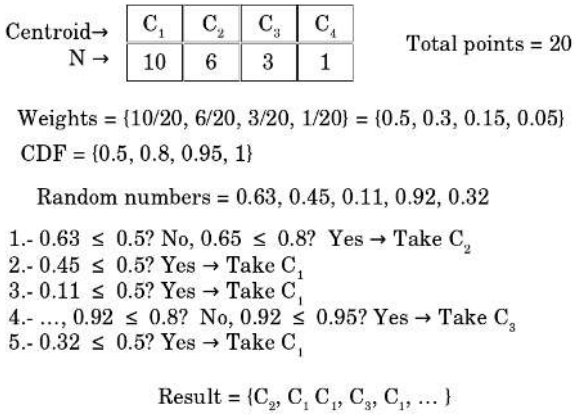


Figure 3.7.: Demonstration: sampling the centroids from weights

4. Analysis and results

This chapter aims to expose an analysis of the results for the adaptation of the *CluStream* method in Spark. First, validation tests were performed in order to determine whether this adaptation compares to what the original paper describes it should. This test focuses only on the errors compared to the original version. Secondly, a thorough analysis on its performance and scalability is presented. These tests are done in a cluster set up specifically for Spark in the LRZ Cloud service. Finally, this adaptation is compared against some other methods available for Spark, regarding stream clustering: *Streaming K-Means* and a related implementation of *CluStream* called *StreamDM*. The adaptation in this project will be referred as *Spark-CluStream*.

4.1. Streaming simulation

All the tests in this section were done using the same streaming mechanism. The dataset in question is contained in a single file that is later divided into several smaller files which are later sent to the streaming receiver in Spark.

The stream receiver in Spark is a text socket receiver, which means that it is expecting text on a predefined TCP socket; for example, *127.0.0.1:9999*, which would be the common *localhost* address and the port *9999*.

To simplify this procedure, a *Bash* script was written, this script reads the files contained in a specified directory, which are named *0,1,2....* These files contain parts of the dataset, so if a file has n points, then n points are sent to the socket for every file. This script benefits from open source software that is available to most *Linux* distributions.

The following code shows how this is done:

```
1  #!/bin/bash
2
3  DIR_STREAM_SPLIT="PATH_TO_THE_FILES_TO_STREAM"
4  TIME=0.25
5  CNT=0
6  while [ -f $DIR_STREAM_SPLIT/$CNT ]; do
7      cat $DIR_STREAM_SPLIT/$CNT | nc localhost 9998
8      echo "Streamed file no. $CNT"
9      let CNT=CNT+1
10     sleep $TIME
11 done
```

This script loops for all the files that are contained in the specified directory, sending each through a socket and waiting $TIME$ seconds in every iteration. By modifying $TIME$, it is possible to control the speed of the stream. For example, if each file contains 100 points, then for $TIME = 0.25$ a listener would receive 400 points every second, for $TIME = 0.1$ a listener would receive 1000 points every second, and so on.

The program *Netcat* is a computer network utility that allows sending and receiving messages using the TCP protocol. In the case of this script, as in line 7, the contents of a file are read and then used as argument for *Netcat* (nc) in order to send them to the *localhost* on the port *9998*. This would mean that the receiver is waiting in that exact address. *Netcat* can also be used to establish connections by calling *nc -lk 9998*, which would create a listener on that port. One thing to notice is that these are one-to-one connections.

4.2. Validation results

Two cases are analyzed in this section. These cases are two of the ones used by the developers of *CluStream* to test its clustering capabilities. The tests differ in two ways: the speed of the stream and the time window used. The desired result would be to obtain comparable results for *Spark-CluStream*.

4.2.1. Setup

These tests were performed in a virtual machine running *Manjaro*, a distribution of *Linux*. The specifications of the virtual machine are:

- *Intel* i7-4500U 64-bit processor @ 1.80-2.40 GHZ.
- Hypervisor vendor: *VMware*.
- Two real cores with hyper-threading for a total of 4 virtual cores.
- 4GB of RAM.
- *Linux-Kernel* version 4.1.

The dataset used for this test is the reduced version of the *Network Intrusion Detector* dataset, which was used for the *Knowledge Discovery and Data Mining Tools Competition* in 1999[1]. The same dataset was used for in the original paper when the *CluStream* method was published[8] in order to test the accuracy of it and to compare it to an older method *STREAM*. This dataset has nearly 500000 entries with a total of 42 attributes each, from which 34 are numerical. Only the numerical attributes are being used in this test and the goal is to find 5 clusters.

4.2.2. Case 1

The first case uses a stream speed of 2000 points per time unit and a horizon $H = 1$. The dataset contains exactly 494021 points, meaning that the online phase would require $\frac{494021}{2000} \approx 247$ time units to complete.

The measurement is the sum of squares (SSQ) of the euclidean distances from the points to their nearest macro-cluster. In fact, the case is run a total of 4 times for *Spark-CluStream* to compute an average. The SSQ is defined as:

$$SSQ = \sum_{i=1}^{N} distance(p_i, c)^2, \tag{4.1}$$

[1]More information: https://archive.ics.uci.edu/ml/datasets/KDD+Cup+1999+Data

where N is the number of points used in the horizon $H = 1$, which should average $N \approx 2000 \cdot H \approx 2000$ points.

The parameters used in [8] are: $\alpha = 2, l = 10, InitNumber = 2000, \delta = 512, t = 2$.

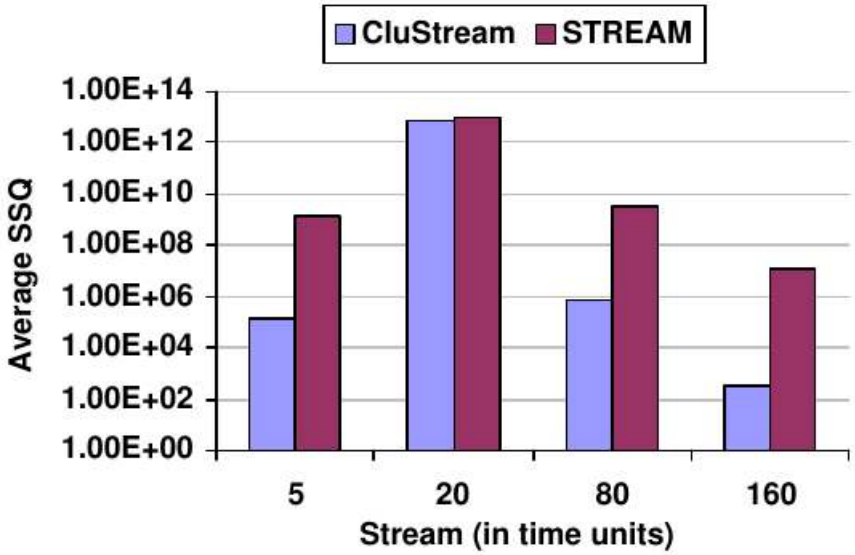


Figure 4.1.: Results for the original $CluStream$[8]. Stream speed = 2000, H=1

Figure 4.1 shows the results used by the original *CluStream* to show its capabilities against an older method *STREAM*, which is a modified version of K-Means for data streams. The average SSQ for *CluStream* is the most relevant to this test.

The parameters used for *Spark-CluStream* were matched.

The parameter m, for m last points, was the only one not provided. Here, $m = 20$ was chosen. For this case, both m and δ are irrelevant and the reason is that the threshold is never reached (247 time units vs. 512). The number of micro-clusters q is 50, a 10 times the number of final clusters (5) is enough for the vast majority of cases[8]. The rest of the parameters were matched, with the only remaining thing to point out is that *fakeKMeans()* used 5000 sampled points.

Figure 4.2 shows the results obtained by *Spark-CluStream*. There is a difference in the labels of the horizontal axis, while Figure 4.1 shows the time units of the stream, Figure 4.1 shows the number of points that had been streamed and processed.

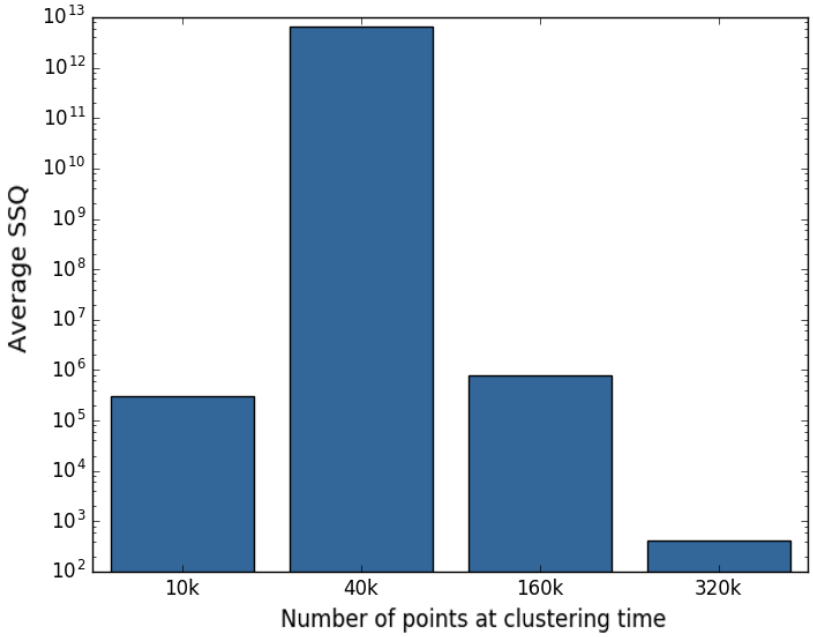


Figure 4.2.: Validation results for $Spark - CluStream$. Stream speed = 2000, H=1

This is done because Spark streaming libraries in combination with the streaming simulation do not always deliver the same amount of points every single time unit, leading to inaccurate results comparing only by clustering on certain time units. A basic multiplication was used to determine the exact moment in terms of points: $2000 \cdot 5 = 10000$, $2000 \cdot 20 = 40000$ and so on.

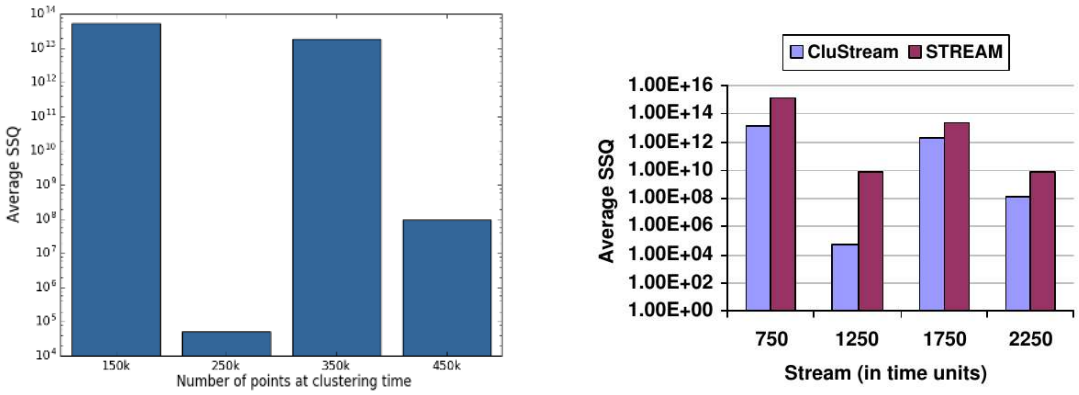
Comparing the results, it is possible to deduce that they are very similar. The exact values for Figure 4.1 are not available but it suffices to compare the magnitudes of the average SSQ.

Case 1: SSQ	**10k**	**40k**	**160k**	**320k**
CluStream	10^5-10^6	10^{12}-10^{13}	$\approx 10^6$	10^2-10^3
Spark-CluStream	3.099×10^5	6.676×10^{12}	7.833×10^5	4.191×10^2

4.2.3. Case 2

The second case uses a stream speed of 200 points per time unit and a horizon $H = 256$. The dataset contains exactly 494021 points, meaning that the online phase would require $\frac{494021}{200} \approx 2470$ time units to complete.

The measurement is the SSQ again and the same circumstances apply for this case as in the first one with the difference that here δ and m are relevant. The parameter m is again chosen to be 20: if 200 points are processed every time unit and there are 50 micro-clusters, assuming all 200 points should be distributed uniformly at least every 5 time units leads to $5\frac{200}{50} = 20$. An in-depth analysis of the behavior of *CluStream* for different δ's and m's is out of the scope of this work.



(a) *Spark-CluStream* (b) Original *CluStream*[8]

Figure 4.3.: Validation results: case 2. Stream speed = 200, H = 256

Again, the comparison is for the average SSQ. The test ran 4 times for *Spark-CluStream* to average the results, which are very similar to the original *CluStream* in this case as well:

Case 2: SSQ	**150k**	**250k**	**350k**	**450k**
CluStream	10^{13}-10^{14}	$\approx 10^5$	10^{12}-10^{13}	$\approx 10^8$
Spark-CluStream	5.402×10^{13}	5.143×10^4	1.892×10^{13}	9.646×10^7

4.3. Performance and optimization analysis

This section intends to explore the performance capabilities of *Spark-CluStream*. One of the purposes of this project is to incorporate an advanced stream clustering method in Spark, which is a platform for distributed computing, meaning that it is important to know how it can be benefited from parallel computations. Scalability in different scenarios is the main focus of these tests.

4.3.1. Setup

In order to test the performance of *Spark-CluStream* using different numbers of processors, a stand-alone Spark cluster was set up in the *LRZ Compute Cloud* service, which is part of the *Leibniz Supercomputing Center*. The specifications are:

Spark cluster - LRZ Cloud		
	Per VM	**Total (5 VM's)**
O.S.	*Ubuntu Linux*	
CPU	*Intel Xeon E5540 @ 2.53GHz 64-bit*	
Hypervisor	KVM	
Cores	8	40
Threads/core	1	1
RAM (GB)	16	80

Two new datasets are used in addition to the *Network Intrusion* dataset. These are two randomly generated datasets using radial basis functions for 2 and 100 dimensions, respectively. Each dataset contains 10000 points, from which 1% is noise. The randomly generated points correspond to two clusters that drift over time. These datasets are used to provide a controlled amount of noise and therefore, the number of outliers for the *CluStream* method remains as constant as possible during the online phase. The quality of the clustering for these datasets is not tested, they are only used to measure performance. These points where streamed over and over until reaching a total of 490000 points.

4.3.2. Scalablity

The scalability tests are performed in two different scenarios: one being an analysis of how it scales for different number of attributes (dimensions of the data points) using only 20 micro-clusters and the other one using 200 micro-clusters. The reason behind this is that the number of attributes and the number of final clusters for a specific purpose are two key factors which determine the complexity of *SparkCluStream*. The speed of the stream is controlled for 10000 points for every batch of data because it is easier to test the scalability when many computations have to be done.

Any application using Spark streaming assigns one core exclusively to handle the stream, therefore the minimum number of processors required is two, this also means that using 2 processors is equivalent to using a single processor to execute the application. The number of processors mentioned in these tests is the total, but the real number of processors used for the computations is that number minus one.

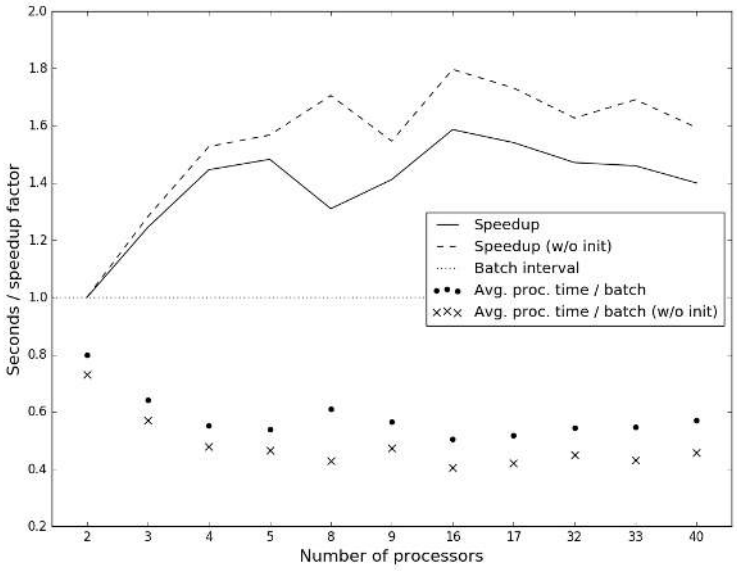


Figure 4.4.: Scalability: Stream speed = 10000, q = 20, d = 2

The charts here presented show the speedup obtained by increasing the number of processors from 2 to 40, which in reality means that 1 to 39 processors where used for the computations. It also shows the average processing time for each batch of data. Because the initialization takes the most amount of time, it is also convenient to show these values without considering that process: by doing so it is possible to see what would be the expected results for a longer run, where the initialization is no longer dominant. Finally it shows the interval time for which Spark processes a new batch of data, in particular all these tests processed one batch every second.

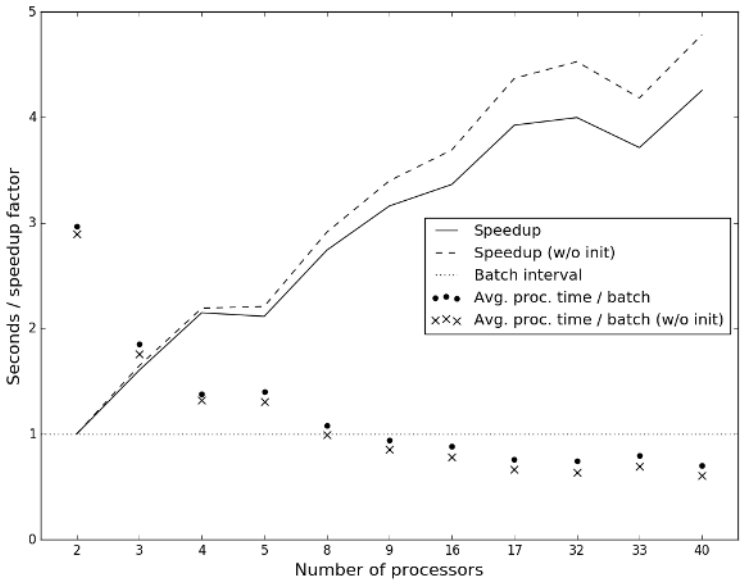


Figure 4.5.: Scalability: Stream speed = 10000, q = 20, d = 100

Figure 4.4 shows that using only 20 micro-clusters and 2 dimensions has poor scalability, not even being able to perform twice as fast as for a single processor (2 in total). Even for this high speed streaming, one processor is enough to process the batches of data before a new batch is processed, meaning that the setup is stable.

Increasing the dimensionality of the points increases the computational effort

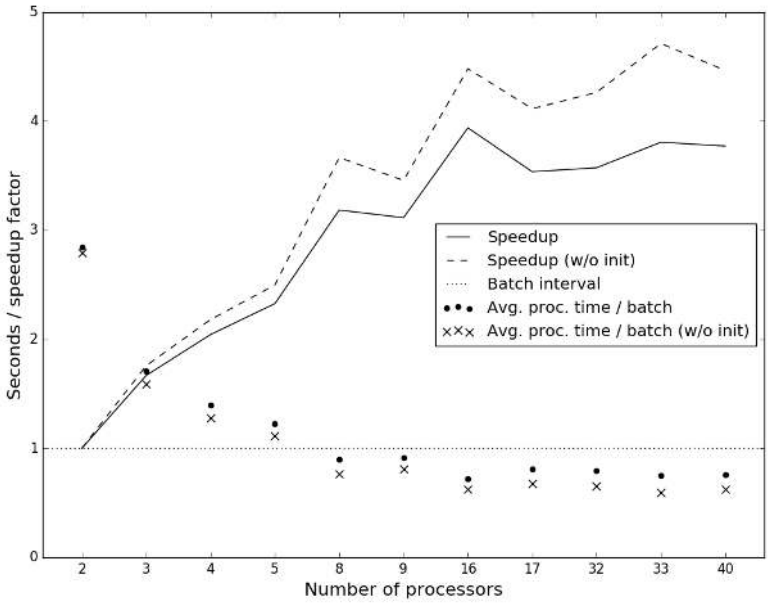


Figure 4.6.: Scalability: Stream speed = 10000, q = 200, d = 2

needed to process the points in every batch of data and here is where *Spark-CluStream* shows its scalability, which is almost linear[2] for up to 16-17 processors, as it can be seen in Figure 4.5. From the average processing time per batch, it can be seen that from 32 to 40 processors it does not improve much anymore and the speedup does not increase quasi-linearly anymore. Here a total of 9 processors were required to stabilize *Spark-CluStream*.

Interestingly, increasing the number of micro-clusters by a factor of 10 for 2 attributes resulted in good scalability, similarly to the scenario with 20 micro-clusters and 100 attributes. Here a total of 8 processors were enough for a stable run, as shown in Figure 4.6.

Finally, when the number of clusters and the number of attributes are both increased significantly, Figure 4.7 shows for *Spark-CluStream* quasi-linear scalability but this time only up to about 8-9 processors. After that point, the speedup slows

[2]By linear scalability does not mean it scales with a 1 to 1 ratio, but rather linearly proportional.

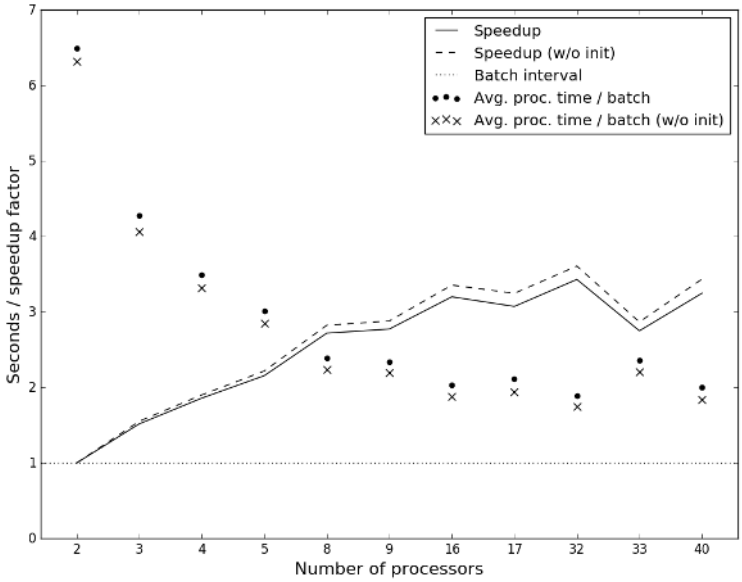


Figure 4.7.: Scalability: Stream speed = 10000, q = 200, d = 100

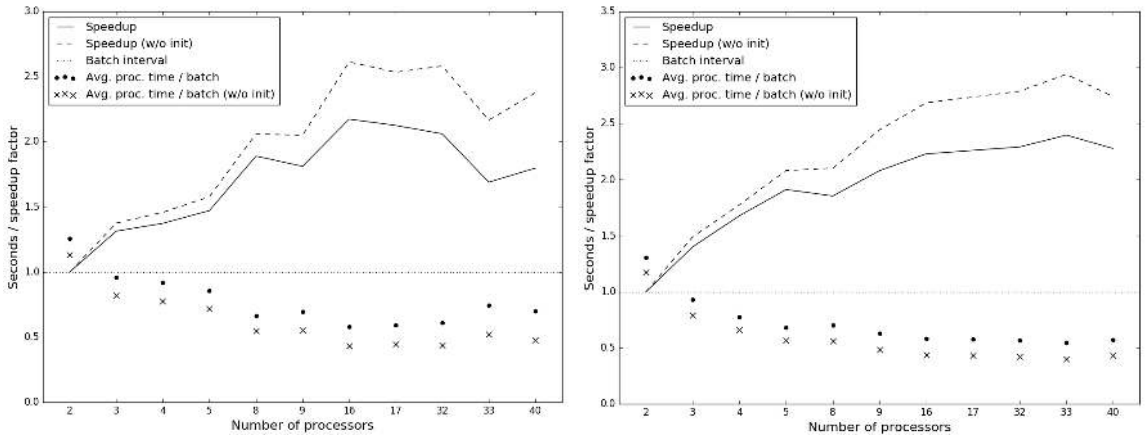
down showing almost no improvement after 16 processors. This test never reached a stable configuration.

4.3.3. Optimization and performance notes

Another remark is that these tests were performed using the same number of partitions than the number of real processors used for computation. The number of partitions refers to the number of parts the DSstream is divided at the time Spark distributes the data. This is a common good practice while tuning Spark applications.

To demonstrate the effect of manually setting the number of partitions to use compared to the default partitioning strategy, a simple test was performed using the *Network Intrusion dataset*, which would be a realistic scenario.

Figure 4.8 shows that not only the scalability is more consistent when the number of partitions is equal to the number of processors, but *Spark-CluStream* shows



(a) Default partitioning (b) Num. partitions = num. processors

Figure 4.8.: Scalability: partitioning strategies. Stream speed = 1000

better scalability.

Noticeable bottlenecks

Every parallelized algorithm is subject to communication costs. Spark applications are no exception, when there is a *reduce* operation, or any operation that requires collecting data, Spark has to communicate and operate the distributed data. In *Spark-CluStream*, micro-clusters are maintained locally, meaning that all the parallel computations are performed and then the results are transferred so the micro-clusters can be updated and the outliers handled. In Figure 4.4 it becomes evident that after a total of 16 processors, the speedup not only stops increasing but starts decreasing. As that test was not particularly computationally intensive, it stopped benefiting from more processing units and stared being affected by the communication times required to send and receive all the information from many different nodes.

Another obvious bottleneck is the complexity of handling the outliers. This operation increases noticeably when the number of micro-clusters is increased, being affected by the number of attributes and the number of outliers processed as well.

Assuming that 1% of the points in a batch are outliers which require a merge, an approximation of the operations (of the most expensive algorithms) needed for processing the points, $O((points) * d * q)$, and handling the outliers, $O((outliers) * d * \frac{q!}{2!(q-2)!})$, is shown using the values of the previous tests:

1. $q = 20, d = 2 \rightarrow O(0.99(points) * 2 * 20) = 39.6(points), O(0.01(points) * 2\frac{20!}{2!(18)!}) = 3.8(points)$ and the ratio is $\frac{39.6}{3.8} \approx 10.42$
2. $q = 20, d = 100 \rightarrow O(0.99(points) * 100 * 20) = 1980(points), O(0.01(points) * 100\frac{20!}{2!(18)!}) = 190(points)$ and the ratio is $\frac{1980}{190} \approx 10.42$
3. $q = 200, d = 2 \rightarrow O(0.99(points) * 2 * 200) = 396(points), O(0.01(points) * 2\frac{200!}{2!(198)!}) = 398(points)$ and the ratio is $\frac{396}{398} \approx 0.99$
4. $q = 200, d = 100 \rightarrow O(0.99(points)*100*200) = 19800(points), O(0.01(points)* 100\frac{200!}{2!(198)!}) = 19900(points)$ and the ratio is $\frac{19800}{19900} \approx 0.99$

It is clear that the more micro-clusters have to be processed, the more *Spark-CluStream* suffers from handling the outliers, that is why Figure 4.7 shows the worst performance among all tests.

Even though the ratio between the parallelizable operations (processing the points) and sequential operations (handling outliers) indicates that the test for $q = 20, d = 100$ should be more scalable than the one for $q = 200, d = 2$, it is important to consider, again, communication costs, for example: each micro-cluster contain two vectors of dimension d, meaning that for the test when $q = 20$ and $d = 100$, $20 * 1000 = 2000$ values (*doubles*) for each vector have to be communicated repeatedly, in contrast to only $200 * 2 = 400$ for the other case.

Implementation considerations

There is one particular remark regarding the code with respect to parallelization in Spark that potentially improves the performance significantly. In section 3.3.3 (Processig points), it is stated that every point is mapped to obtain a tuple containing the point itself, its elements squared and the number 1 such that the reduction

of all points would get the sum of the values of that tuple all at once. At some point, those operations were done separately, getting one RDD for the sum of the points, another RDD for the sum of the squared values and another RDD for the count of points:

```
1 val pointCount = assignations.groupByKey.mapValues(a => a.size).collect
2 val sums = assignations.reduceByKey(_ :+ _).collect
3 val sumsSquares = assignations.mapValues(a => a :* a).reduceByKey(_ :+ _
    ).collect
```

Doing those operations separately means that the RDD *assignations* has to be reduced and collected three times instead of once, resulting in more communication costs and therefore slowing the algorithm down.

4.4. Comparison against alternatives

It is important for this project to know how *Spark-CluStream* stands against some of the other alternatives for stream clustering available for Spark, in particular: *Streaming K-Means* from Spark and *StreamDM-CluStream*, which is another adaptation of the *CluStream* method for Spark. There are two aspects of interest in this tests, one being their clustering capabilities and the other their performance.

4.4.1. Clustering

The setup and the dataset are the same as in 4.2, as having already verified results provides the possibility of using those tests to directly compare the results against the other methods. Again, the used measurement is the sum of squares (SSQ).

Before looking at the results, here are some key considerations for the other methods:

- *Streaming K-Means*:
  - In order to have comparable results, the time horizon H must be interpreted differently. There are two strategies: the first option is to use

the parameter *halfLife*, which can be configured to let the algorithm to completely adjust the clusters after HL points or batches.

  - The alternative would be to set the $decayFactor$, which sets the weight for the clusters of the "old" data (only the current batch is considered "new" data). This is a number between 0 and 1, such that if it is 0 then only the clusters for "new" data determine the final clusters, if it is set to 1, then the clusters of past data will have the same influence on the final clusters. It is important to notice that this *decayFactor* also considers the number of points of the "new" and "old" data, so in the last case, after a long time, "new" data will have little influence as the number of points of the current batch will be considerable smaller than the points clustered so far.

- *StreamDM-CluStream*:
  - This adaptation of *CluStream* does not include the offline part as a separate module, meaning that it does not save snapshots and therefore it has to perform the macro-clustering process for every batch. This brings some limitations, the horizon H no longer has the same meaning: the δ parameter is used instead as an equivalent, relying on the micro-clustering part only and its ability to delete and create new micro-clusters.

Case 1

The parameters used for *Spark-CluStream* are the same as in 4.2. The number of clusters k is always 5 for this dataset and these tests for all methods.

For *Streaming K-Means*, the horizon $H = 1$ was transformed to $halfLife = 1000$ points. This is because the speed of the stream is 2000 points per time unit, if the horizon is 1, then only 2000 points are desired to be clustered, and half of that results in 1000 points. For the *decayFactor*, it is safe to choose 0, as that would mean that only the last 2000 points have influence on the clusters, which is exactly what it's desired.

StreamDM-CluStream is set up with its default parameters, only changing the horizon to 1 and the number of micro-clusters to 50 in order to match those of *Spark-CluStream*.

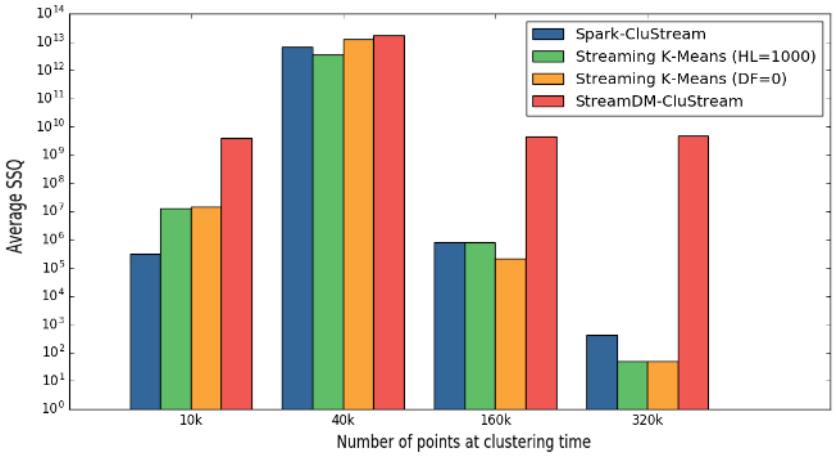


Figure 4.9.: Comparison results: all methods. Stream speed = 2000, H=1

From Figure 4.9 it can be seen that *Spark-CluStream* delivers results which are very close to those of *Streaming K-Means*, which performs significantly better than the older method *STREAM*. Also, *Streaming K-Means* with the *decayFactor* (DF) is expected to do well on this test as it could be configured to cluster exactly as it was intended for this dataset.

The surprising results came from *StreamDM-CluStream*, as it performed noticeably, and significantly, worse than the rest of the methods. Specially for the last two marks at $160k$ and $320k$ it shows poor performance, which are where the other methods performed the better on average.

To find out whether this behavior is due to not using the snapshots plus offline macro-clustering, another test was performed using *Spark-CluStream* with the same conditions as for *StreamDM-CluStream*: using $\delta = 1$ as the horizon and $m = 100$ to match both methods

Figure 4.10 shows poorer results for *Spark-CluStream* in comparison to its original behavior with snapshots, but still delivers noticeably better results than *StreamDM-*

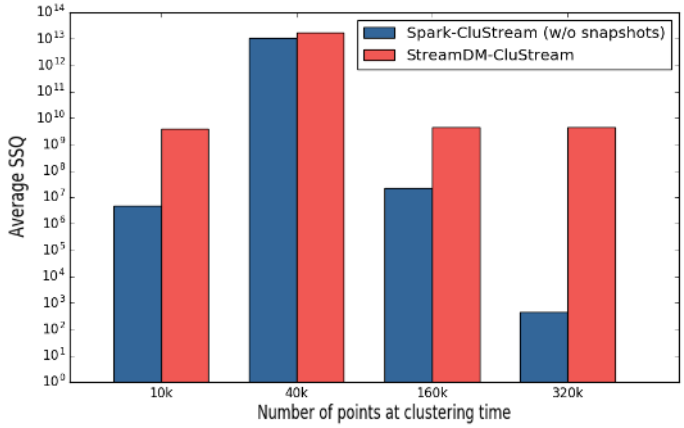


Figure 4.10.: *Spark-CluStream* without snapshots. Stream speed=2000, H=1, m=100

CluStream, even though all these tests were executed 4 times and the SSQ erros were averaged to get a better representation of how these methods perform.

Case 2

Repeating the experiment for the stream with a speed of 200 and a horizon $H = 256$ revealed unexpected results. While most parameters for all methods remained the same, for *Streaming K-Means* a new $halfLife$ has to be calculated: multiplying the speed of the stream to the horizon, $200 \cdot 256 = 51200$ shows how many points of the stream are supposed to be clustered at each time, indicating that the parameter should be set to $halfLife = 25600$.

The *decayFactor* strategy at first seems that does not work for such experiment, but considering that the total number of entries is known and exactly the marks at which the clustering process happens, it is possible to calculate an average value to use as a $decayFactor$:

- At 150000 points: $\frac{51200}{150000} \approx 0.3413$, which is the ratio of the points to cluster to the total number of points at that particular time.
- At 150000 points: $\frac{51200}{250000} \approx 0.2048$.

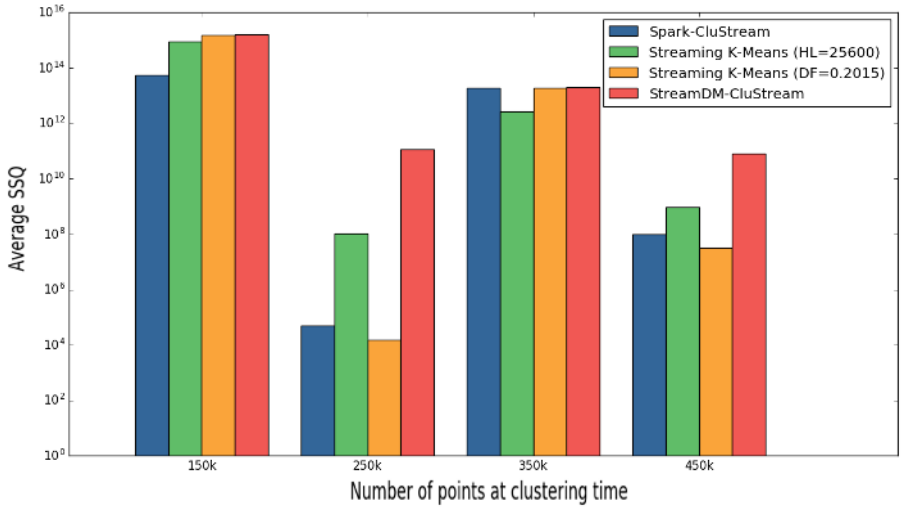


Figure 4.11.: Comparison results: all methods. Stream speed = 200, H=256

- At 150000 points: $\frac{51200}{350000} \approx 0.1462$.

- At 150000 points: $\frac{51200}{450000} \approx 0.1137$.

Averaging those ratios leads to a *decayFactor = 0.2015*, which is a way to determine how important the old data is in comparison to the new one.

Figure 4.11 shows that while *Spark-CluStream* still performs consistently good, *Streaming K-Means* with the *decayFactor* outperformed its relative with the $halfLife$ strategy. Another thing to notice is that *StreamDM-CluStream* still delivered the worse results.

Testing *Spark-CluStream* again without the use of snapshots, showed once more that it delivers better results than *StreamDM-CluStream*, as it can be seen in Figure 4.12, but the difference was reduced significantly. These results might indicate that *StreamDM-CluStream* does not benefit from shorter horizons.

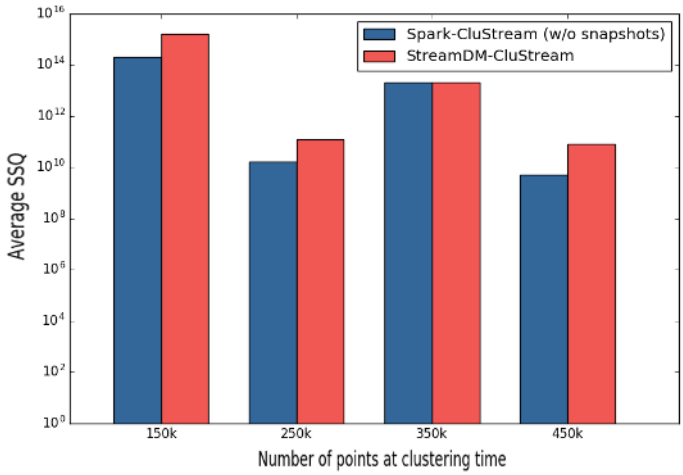


Figure 4.12.: *Spark-CluStream* without snapshots. Stream speed = 200, H=256, m=100

4.4.2. Performance

In this section, the scalability of *Spark-CluStream* is compared to that of *StreamDM-CluStream* and Spark's *Streaming K-Means* unsing the Spark cluster setup for $q = 20$ and $d = 2,100$, for the $CluStream$ method. Also, a test on a signle machine is performed, using the setup and dataset as in 4.2.

In Figure 4.13 it can be seen that *Spark-CluStream* took the most time on average to process a batch of data and being *Streaming K-Means* the fastets among the three.

When it comes to higher dimensions, *Spark-CluStream* shows a significant improvement over *StreamDM-CluStream*, which never got to the point were it was stable (below the 1 second mark), as shown in 4.14, it seems to scale as fast as *Spark-CluStream* but it was not enough even with 40 processors.

Surprisingly, in Figure 4.15, *StreamDM-CluStream* shows to be able to scale even for this tests, while both *Spark-CluStream* and *Streaming K-Means* seem to struggle taking advantage of using more processors.

Figure 4.16 shows that all three algorithms are able to scale similarly for this test, being *Spark-CluStream* the one having a very slight advantage as it does not slow

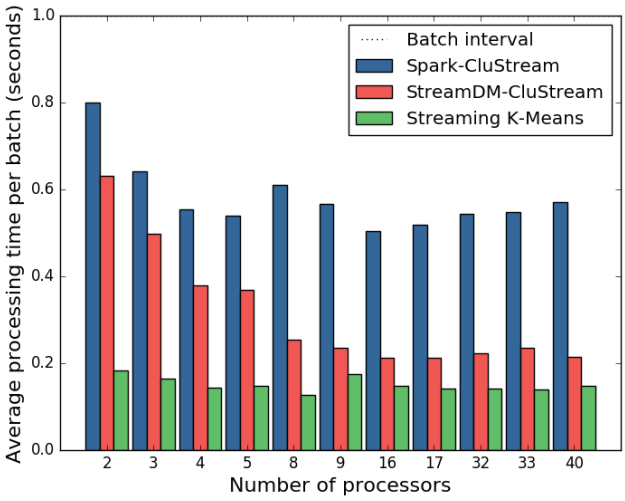


Figure 4.13.: Processing time comparison: $q = 20$, $d = 2$

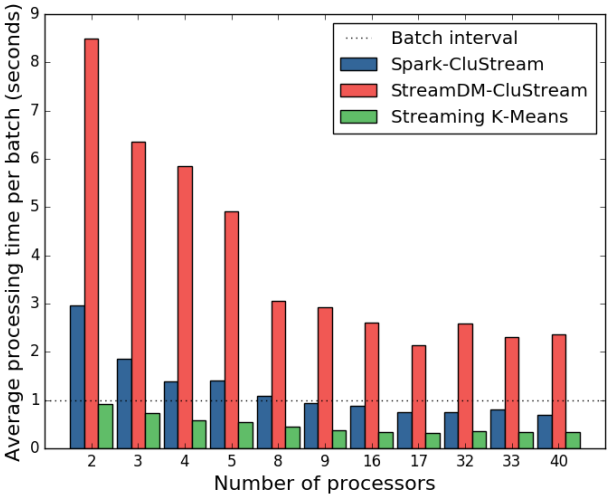


Figure 4.14.: Processing time comparison: $q = 20$, $d = 100$

down as quickly as the other two.

Another interesting comparison, is the processing time per batch of data for a single machine, using a real dataset such as the *Network Intrusion*. Here, communication is less of an issue as all the partitions lie in the same share memory space, and still there are 4 virtual cores in disposition for the algorithms to run.

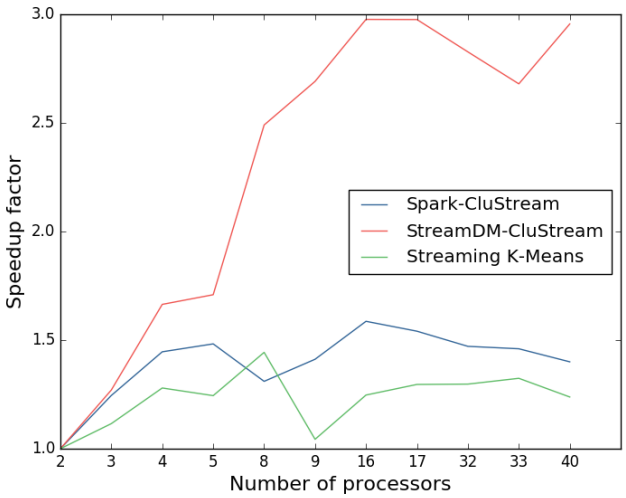


Figure 4.15.: Scalability comparison: $q = 20$, $d = 2$

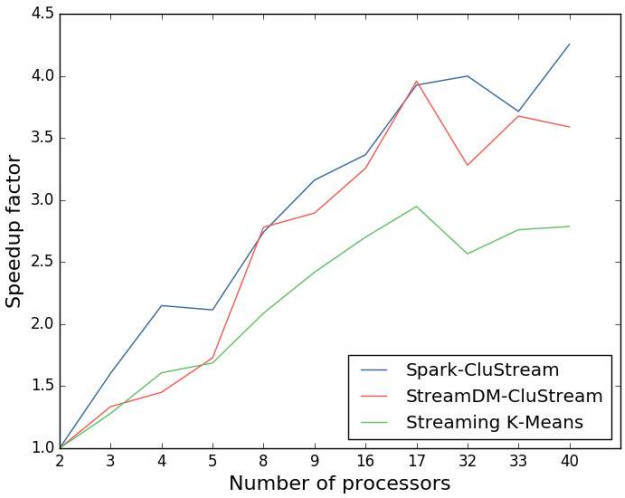


Figure 4.16.: Processing time comparison: $q = 20$, $d = 100$

The test was performed using a stream speed of 2000 points per batch and with a horizon $H = 1$, to match one of the validation tests.

The results shown in Figure 4.17 are quite remarkable. As *StreamDM-CluStream* shows a very significant disadvantage when using greater numbers of micro-clusters

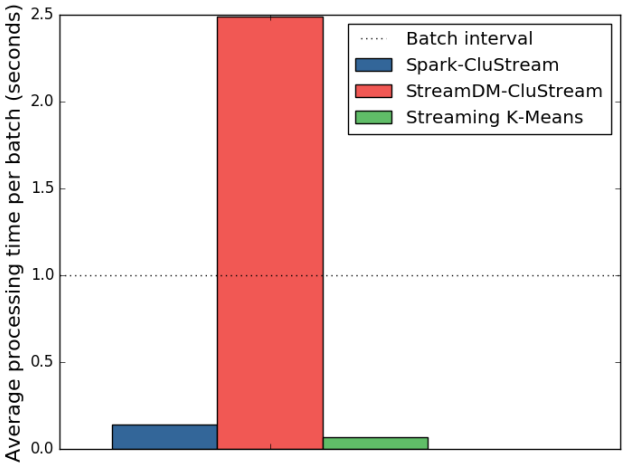


Figure 4.17.: Processing time comparison for a single machine: $q = 50$, $d = 34$

and higher dimensions.

For this single machine test, *Spark-CluStream* was about 18 times faster on average than *StreamDM-CluStream* and about two times slower than *Streaming K-Means* on average.

Another consideration to be made, is that *Spark-CluStream* saves a snapshot for every batch of data, having to write to disk, while the other two algorithms never access the disk or this matter.

5. Conclusions

Bringing successfully the *CluStream* method to Spark has provided valuable information about how similar methods could be adapted by reviewing some of the challenges which might occur. It has also provided deep understanding of this method itself and how stream clustering differs from other clustering methods, which might not need to adapt to changing data, of Apache Spark and how distributed systems work in general, but most importantly it has provided the experience to parallelize algorithms for the specific requirements of a given problem and this knowledge itself is applicable to an uncountable number of problems.

The typical workflow for doing so should follow a similar structure: from understanding the problem as described in chapter 2.3.2, to going through the process of achieving the desired goals as shown in chapters 3 and 4.

5.1. Goals review

It is rewarding to conclude that the goals were met satisfactorily. Here, the conclusions for every one of them.

5.1.1. Adapt *CluStream* in Spark (Spark-CluStream)

This is the main goal of this thesis, and none of the other goals would have been met if this one failed. Adapting *CluStream* in Spark brought many challenges: one being the fact that the streaming library in Spark handles streams as batches and not individual points with time stamps, forcing this adaptation to change a few things that differentiate it from the original method, and the other one being the

parallelization of the algorithm in order to take advantage of distributed computing.

Even though this adaptation changed the way data is processed, i.e. processing the stream in batches of data in parallel as much as possible, the results indicate that this was done correctly and the proof of that lies in the validation section 4.2: it showed that it is not only capable of correctly clustering streams of data but it was able to match the quality of the original method described in [8]. It was shown for two different scenarios that this is true, when comparing the errors obtained after replicating the tests done by the authors of *CluStream*.

5.1.2. Understanding its advantages and disadvantages

The second most important goal was to make this adaptation as scalable as possible, and for this reason many tests were made using different scenarios. There are clearly cases where it is not fully scalable, as shown in section 4.3, but for the most part it was shown comparable scalability as some of the alternatives for Spark, including a method native of Spark and a similar method developed by a team from *Huawei* for a set of stream mining algorithms called *StreamDM*.

Some of its limitations were also understood , such as bottlenecks that might reduce the scalability and performance in general, such as:

- Outliers: handling with outliers in sequential code is expected to be a bottleneck, depending on the stream, a batch of data might contain points which do not belong to any micro-cluster and therefore, they have to be handled differently. Depending on the number of outliers, in particular the ones which require two micro-clusters to be merged, the total processing time for that batch will be affected negatively. In general there are three situations where this would normally occur: at the beginning of the stream if the initialization is not accurate, when the incoming data is very noisy and when the data dramatically changes.

- Communication costs: running in parallel requires certain communication between processing units. This affects the scalability negatively when few

computations are required and too many processing units are used, as most of the time will be spent on communication. Also, as shown in the chapter section 4.3, even when it is expected to be scalable, increasing the number of micro-clusters used and the dimensionality of the data results in a bigger amount of information to communicate, and therefore not allowing greater speedups after a certain amount of processing units.

The results also showed that the *Streaming K-Means* algorithm is the fastest among the three tested (highly optimized for Spark), delivering good results in certain scenarios as it does not count with the flexibility of *CluStream* to better fit to evolving streams. *Spark-CluStream* on the other hand, showed that it not only delivers quality clustering, but also outperformed the similar *CluStream* implementation in *StreamDM*. Quality-wise it delivered more consistent and accurate results, and performance-wise it outperformed it in most cases, including one up to around 18 times faster.

5.1.3. Contribute to the Apache Spark project

The Spark project, as part of the Apache Software Foundation, is an open source project with great potential, being used more and more in the past few years. It seems only logical to contribute to this project after developing something for it. Having successfully achieved the goals makes it a worthy contribution that has this thesis supporting it.

There is a collection of packages that integrate seamlessly with Spark. This collection can be found in *spark-packages.org*. There it is also possible to find the *Spark-CluStream* package and it is available to anyone using Spark. That package is linked to a *Github* repository for anyone who desires to explore, download and modify the source code, these can be found here:

- Spark package: http://spark-packages.org/package/obackhoff/Spark-CluStream
- Repository: https://github.com/obackhoff/Spark-CluStream

5.1.4. Final words

After successfully adapting the *CluStream* method in Apache Spark, validating its qualities and discovering its strengths and weaknesses, contributing to an open source project, as big as this one, is a very pleasing way to conclude this thesis, with many lessons learned and many challenges to come.

Appendix

A. Algorithms

A.1. K-Means

Description

The K-means algorithm is one of the simplest algorithms that solve the clustering problem. The idea behind it is to start with a random guess of where the K centers of the clusters will be, then assigning every point to its closest center and finally recompute the centers by averaging the the points (vectors) contained each cluster. This process is repeated until convergence, i.e. when points no longer change of cluster.

Consider the following objective function[10]:

$$J = \sum_{n=1}^{N} \sum_{k=1}^{K} r_{nk} ||x_n - c_k||^2$$

where $x_n \in \{x_1, x_2, ..., x_N\} | x_n \in \mathbb{R}^d$ is an element from the set of points, N is the number of entries of the dataset or number of points and d is the dimension of each point; $c_k \in \{c_1, c_2, ..., c_K\}$ is a point that represents the center of a cluster. The term $r_n k \in \{0, 1\}$ is an indicator function such that for the point x_n and the center c_k is equal to 1 if the point is assigned to that cluster and 0 otherwise.

It consists of two optimization steps for the objective function:

1. For $r_n k$ we simply minimize it by assign it to $\begin{cases} 1 & \text{if } k = argmin_j ||x_n - c_k||^2 \\ 0 & \text{otherwise} \end{cases}$

2. For c_k we minimize the objective function setting its derivative equal to zero:

$$2\sum_{n=1}^{N} r_{nk}(x_n - c_k) = 0$$

which we can solve

$$c_k = \frac{\sum_n r_{nk} x_n}{\sum_n r_{nk}} \tag{A.1}$$

These two steps are repeated until convergence, which can be defined as when the points do not change clusters anymore. Also, a maximum number of iterations or another stopping criterion can be defined.

The MapReduce approach

To implement the K-means algorithm with a MapReduce model it is necessary to divide the algorithm in such a way that the sub-tasks can be computed in parallel. First, we consider a *map* function that will take an *object* of the dataset and calculate its distance to the *centroids* of the clusters, returning the *nearest centroid* and the *object* itself. Doing this is equivalent to assigning the *object* to the cluster with that resulting *nearest centroid*. This is illustrated in Figure A.1.

The *map* function described in the Algorithm 14 runs on each *mapper* and uses *distance(a,b)* function which computes the distance between two objects *a*, *b*, i.e. using the *Minkowski* distance:

$$\sqrt[q]{\sum_{i=1}^{d} |a_i - b_i|^q} \tag{A.2}$$

where q is usually set to 2 and being exactly the $Euclidean$ distance.

The $reduce$ function is described in the Algorithm 15 and, similarly, it runs on each reducer and uses a function *calculateCentroid(x)* which averages all the objects' coordinates to find the *new centroid* as described in the equation A.1.

Algorithm 14 K-means map function

Input: *centroids, input* — *centroids* is given by the JobConf of MapReduce and contains the previously calculated *centroids*. For the input the *key* is the index of the input segment of the dataset and the *value* is the *object* that is getting clustered.

Output: *output* — The *key* is the nearest centroid to the *object* and the *value* is the *object* itself.

```
1: nearestCentroid ← NULL
2: minDistance ← ∞
3: dist ← NULL
4: for all c ∈ centroids do
5:     dist ← distance(c, input.value)
6:     if nearestCentroid == NULL‖dist < minDistance then
7:         nearestCentroid ← c
8:         minDistance ← dist
9:     end if
10: end for
11: output.collect(nearestCentroid, input.value)
```

Algorithm 15 K-means reduce function

Input: *input* — The *key* is the centroid of the cluster and the *value* is the *list* of objects assigned to the cluster.

Output: *output* — The *key* is the the old centroid and the *value* is the *new centroid*.

```
1: objects ← NULL
2: for all o ∈ input.value do
3:     objects.append(o)
4: end for
5: newCentroid ← calculateCentroid(objects)
6: output.collect(input.key, newCentroid)
```

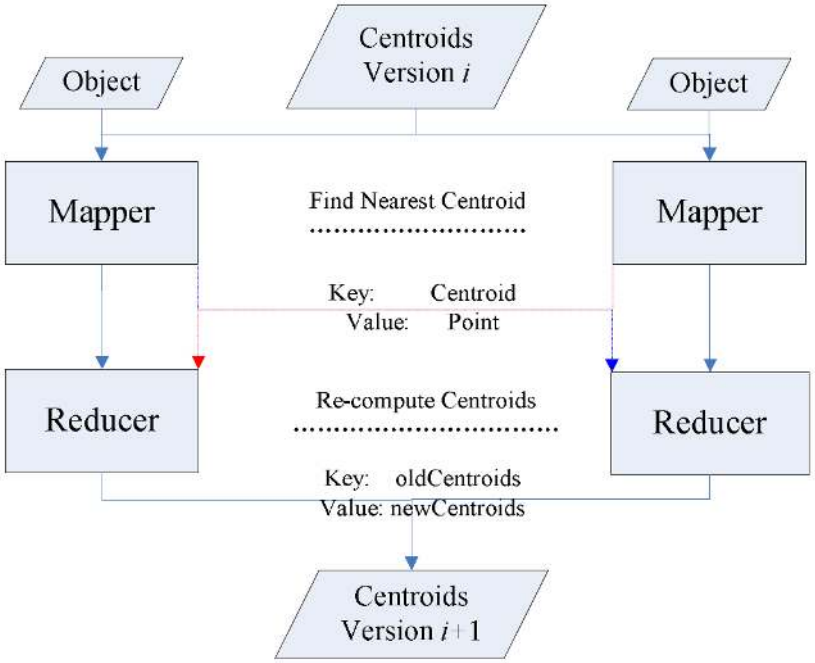


Figure A.1.: K-means with MapReduce. From: [15]

A.2. Update micro-clusters information

To update the micro-clusters information, the *cluster feature vector* is used, but first the tuple is updated using the MicroClusterInfo instance and the first unique ID of the micro-cluster it refers to, this is done in order to identify the micro-cluster at the time of permorfimg parallel computations. For the centroid, the vector $\overline{CF1^x}$ is divided element-wise by the number of points the micro-cluster has. The number of points is copied from the micro-cluster and the RMSD is calculated whenever there is more than one point in the micro-cluster; a separate loop is required when the micro-cluster only has one point as it needs the up-to-date $mcInfo$ array to compute the distance to its nearest micro-cluster. This procedure is shown in Algorithm 16

Algorithm 16 Update mcInfo ($updateMicroClusterInfo()$)

1: $i \leftarrow 0$
2: **for all** $mc \in microClusters$ **do**
3: $mcInfo[i] \leftarrow (mcInfo[i]_1, mc.getIds[0])$
4: $mcInfo[i].setN(mc.getN)$
5: $mcInfo[i].setCentroid(mc.getCf1x :/mc.getN)$
6: **if** $mcInfo[i].getN > 1$ **then**
7: $mcInfo[i].setRmsd(sqrt(sum(mc.getCf2x)/mc.getN - sum(mc.getCf1x.map(a => a * a))/(mc.getN * mc.getN))))$
8: **end if**
9: $i = i + 1$
10: **end for**
11: **for all** $mci \in MicroClusterInfo$ **do**
12: **if** $mci_1 == 1$ **then**
13: $mci_1.setRmsd(distanceNearestMicroCluster(mci_1.centroid, mcInfo))$
14: **end if**
15: **end for**

B. Code

B.1. CluStreamOnline class

```
package com.backhoff.clustream

import breeze.linalg._
import org.apache.spark.broadcast.Broadcast
import org.apache.spark.Logging
import org.apache.spark.streaming.dstream.DStream
import org.apache.spark.rdd.RDD
import org.apache.spark.annotation.Experimental
import org.apache.spark.mllib.clustering.{StreamingKMeans, KMeans}
import breeze.stats.distributions.Gaussian


/**
  * CluStreamOnline is a class that contains all the necessary
  * procedures to initialize and maintain the microclusters
  * required by the CluStream method. This approach is adapted
  * to work with batches of data to match the way Spark Streaming
  * works; meaning that every batch of data is considered to have
  * to have the same time stamp.
  *
  * @param q             : the number of microclusters to use. Normally
    10 * k is a good choice,
  *                       where k is the number of macro clusters
  * @param numDimensions : this sets the number of attributes of the
    data
```

```
  * @param minInitPoints : minimum number of points to use for the
     initialization
  *                       of the microclusters. If set to 0 then initRand
     is used
  *                       insted of initKmeans
  **/

@Experimental
class CluStreamOnline(
                        val q: Int,
                        val numDimensions: Int,
                        val minInitPoints: Int)
  extends Logging with Serializable {


  /**
    * Easy timer function for blocks
    **/

  def timer[R](block: => R): R = {
    val t0 = System.nanoTime()
    val result = block // call-by-name
    val t1 = System.nanoTime()
    logInfo(s"Elapsed time: " + (t1 - t0) / 1000000 + "ms")
    result
  }

  private var mLastPoints = 500
  private var delta = 20
  private var tFactor = 2.0
  private var recursiveOutliersRMSDCheck = true

  private var time: Long = 0L
  private var N: Long = 0L
  private var currentN: Long = 0L
```

```
private var microClusters: Array[MicroCluster] = Array.fill(q)(new
    MicroCluster(Vector.fill[Double](numDimensions)(0.0), Vector.fill[
    Double](numDimensions)(0.0), 0L, 0L, 0L))
private var mcInfo: Array[(MicroClusterInfo, Int)] = null

private var broadcastQ: Broadcast[Int] = null
private var broadcastMCInfo: Broadcast[Array[(MicroClusterInfo, Int)]]
    = null

var initialized = false

private var useNormalKMeans = false
private var strKmeans: StreamingKMeans = null



private var initArr: Array[breeze.linalg.Vector[Double]] = Array()

private def initRand(rdd: RDD[breeze.linalg.Vector[Double]]): Unit =
    {...}

private def initKmeans(rdd: RDD[breeze.linalg.Vector[Double]]): Unit =
     {...}

private def initStreamingKmeans(rdd: RDD[breeze.linalg.Vector[Double
    ]]): Unit = {...}

def run(data: DStream[breeze.linalg.Vector[Double]]): Unit = {...}

def getMicroClusters: Array[MicroCluster] = {...}

def getCurrentTime: Long = {...}

def getTotalPoints: Long = {...}

def setRecursiveOutliersRMSDCheck(ans: Boolean): this.type = {...}
```

```
def setInitNormalKMeans(ans: Boolean): this.type = {...}

def setM(m: Int): this.type = {...}

def setDelta(d: Int): this.type = {...}

def setTFactor(t: Double): this.type = {...}

private def distanceNearestMC(vec: breeze.linalg.Vector[Double], mcs:
    Array[(MicroClusterInfo, Int)]): Double = {...}

private def squaredDistTwoMCArrIdx(idx1: Int, idx2: Int): Double =
    {...}

private def squaredDistPointToMCArrIdx(idx1: Int, point: Vector[Double
    ]): Double = {...}

private def getArrIdxMC(idx0: Int): Int = {...}


private def mergeMicroClusters(idx1: Int, idx2: Int): Unit = {...}

private def addPointMicroClusters(idx1: Int, point: Vector[Double]):
    Unit = {...}

private def replaceMicroCluster(idx: Int, point: Vector[Double]): Unit
     = {...}

private def assignToMicroCluster(rdd: RDD[Vector[Double]], mcInfo:
    Array[(MicroClusterInfo, Int)]): RDD[(Int, Vector[Double])] =
    {...}

private def assignToMicroCluster(rdd: RDD[Vector[Double]]) = {...}

private def updateMicroClusters(assignations: RDD[(Int, Vector[Double
    ])]): Unit = {...}
```

```
}
```

B.1.1. initRand()

```
/**
  * Random initialization of the q microclusters
  *
  * @param rdd : rdd in use from the incoming DStream
  **/

private def initRand(rdd: RDD[breeze.linalg.Vector[Double]]): Unit = {
  mcInfo = Array.fill(q)(new MicroClusterInfo(Vector.fill[Double](
      numDimensions)(rand()), 0.0, 0L)) zip (0 until q)

  val assignations = assignToMicroCluster(rdd, mcInfo)
  updateMicroClusters(assignations)
  var i = 0
  for (mc <- microClusters) {
    mcInfo(i) = (mcInfo(i)._1, mc.getIds(0))
    if (mc.getN > 0) mcInfo(i)._1.setCentroid(mc.cf1x :/ mc.n.toDouble
        )
    mcInfo(i)._1.setN(mc.getN)
    if (mcInfo(i)._1.n > 1) mcInfo(i)._1.setRmsd(scala.math.sqrt(sum(
        mc.cf2x) / mc.n.toDouble - sum(mc.cf1x.map(a => a * a)) / (mc.
        n * mc.n.toDouble)))
    i += 1
  }
  for (mc <- mcInfo) {
    if (mc._1.n == 1)
      mc._1.setRmsd(distanceNearestMC(mc._1.centroid, mcInfo))
  }

  broadcastMCInfo = rdd.context.broadcast(mcInfo)
  initialized = true
}
```

B.1.2. initKMeans()

```
/**
   * Initialization of the q microclusters using the K-Means algorithm
   *
   * @param rdd : rdd in use from the incoming DStream
   **/

  private def initKmeans(rdd: RDD[breeze.linalg.Vector[Double]]): Unit =
      {
    initArr = initArr ++ rdd.collect
    if (initArr.length >= minInitPoints) {
      val tempRDD = rdd.context.parallelize(initArr)
      val trainingSet = tempRDD.map(v => org.apache.spark.mllib.linalg.
          Vectors.dense(v.toArray))
      val clusters = KMeans.train(trainingSet, q, 10)

      mcInfo = Array.fill(q)(new MicroClusterInfo(Vector.fill[Double](
          numDimensions)(0), 0.0, 0L)) zip (0 until q)
      for (i <- clusters.clusterCenters.indices) mcInfo(i)._1.
          setCentroid(DenseVector(clusters.clusterCenters(i).toArray))

      val assignations = assignToMicroCluster(tempRDD, mcInfo)
      updateMicroClusters(assignations)

      var i = 0
      for (mc <- microClusters) {
        mcInfo(i) = (mcInfo(i)._1, mc.getIds(0))
        if (mc.getN > 0) mcInfo(i)._1.setCentroid(mc.cf1x :/ mc.n.
            toDouble)
        mcInfo(i)._1.setN(mc.getN)
        if (mcInfo(i)._1.n > 1) mcInfo(i)._1.setRmsd(scala.math.sqrt(sum
            (mc.cf2x) / mc.n.toDouble - sum(mc.cf1x.map(a => a * a)) / (
            mc.n * mc.n.toDouble)))
        i += 1
      }
      for (mc <- mcInfo) {
        if (mc._1.n == 1)
```

```
            mc._1.setRmsd(distanceNearestMC(mc._1.centroid, mcInfo))
        }

        broadcastMCInfo = rdd.context.broadcast(mcInfo)

        initialized = true
      }
    }
```

B.1.3. initStreamingKmeans()

```
  /**
     * Initialization of the q microclusters using the Streaming K-Means
       algorithm
     *
     * @param rdd : rdd in use from the incoming DStream
     **/
    private def initStreamingKmeans(rdd: RDD[breeze.linalg.Vector[Double
       ]]): Unit = {

     if(strKmeans == null) strKmeans = new StreamingKMeans().setK(q).
        setRandomCenters(numDimensions, 0.0)
     val trainingSet = rdd.map(v => org.apache.spark.mllib.linalg.Vectors
        .dense(v.toArray))

     val clusters = strKmeans.latestModel().update(trainingSet,1.0, "
        batches")
     if(getTotalPoints >= minInitPoints){

       mcInfo = Array.fill(q)(new MicroClusterInfo(Vector.fill[Double](
          numDimensions)(0), 0.0, 0L)) zip (0 until q)
       for (i <- clusters.clusterCenters.indices) mcInfo(i)._1.
          setCentroid(DenseVector(clusters.clusterCenters(i).toArray))

       val assignations = assignToMicroCluster(rdd, mcInfo)
       updateMicroClusters(assignations)
```

```
      var i = 0
      for (mc <- microClusters) {
        mcInfo(i) = (mcInfo(i)._1, mc.getIds(0))
        if (mc.getN > 0) mcInfo(i)._1.setCentroid(mc.cf1x :/ mc.n.
            toDouble)
        mcInfo(i)._1.setN(mc.getN)
        if (mcInfo(i)._1.n > 1) mcInfo(i)._1.setRmsd(scala.math.sqrt(sum
            (mc.cf2x) / mc.n.toDouble - sum(mc.cf1x.map(a => a * a)) / (
            mc.n * mc.n.toDouble)))
        i += 1
      }
      for (mc <- mcInfo) {
        if (mc._1.n == 1)
          mc._1.setRmsd(distanceNearestMC(mc._1.centroid, mcInfo))
      }

      broadcastMCInfo = rdd.context.broadcast(mcInfo)
      initialized = true
    }

  }
```

B.1.4. run()

```
  /**
   * Main method that runs the entire algorithm. This is called every
     time the
   * Streaming context handles a batch.
   *
   * @param data : data coming from the stream. Each entry has to be
     parsed as
   *             breeze.linalg.Vector[Double]
   **/

  def run(data: DStream[breeze.linalg.Vector[Double]]): Unit = {
```

```
    data.foreachRDD { (rdd, timeS) =>
      currentN = rdd.count()
      if (currentN != 0) {
        if (initialized) {

          val assignations = assignToMicroCluster(rdd)
          updateMicroClusters(assignations)

          var i = 0
          for (mc <- microClusters) {
            mcInfo(i) = (mcInfo(i)._1, mc.getIds(0))
            if (mc.getN > 0) mcInfo(i)._1.setCentroid(mc.cf1x :/ mc.n.
                toDouble)
            mcInfo(i)._1.setN(mc.getN)
            if (mcInfo(i)._1.n > 1) mcInfo(i)._1.setRmsd(scala.math.sqrt
                (sum(mc.cf2x) / mc.n.toDouble - sum(mc.cf1x.map(a => a *
                 a)) / (mc.n * mc.n.toDouble)))
            i += 1
          }
          for (mc <- mcInfo) {
            if (mc._1.n == 1)
              mc._1.setRmsd(distanceNearestMC(mc._1.centroid, mcInfo))
          }

          broadcastMCInfo = rdd.context.broadcast(mcInfo)
        } else {
          minInitPoints match {
            case 0 => initRand(rdd)
            case _ => if(useNormalKMeans) initKmeans(rdd) else
                initStreamingKmeans(rdd)
          }
        }
      }
      this.time += 1
      this.N += currentN
    }
  }
```

B.1.5. getMicroClusters()

```
/**
   * Method that returns the current array of microclusters.
   *
   * @return Array[MicroCluster]: current array of microclusters
   **/

  def getMicroClusters: Array[MicroCluster] = {
    this.microClusters
  }
```

B.1.6. getCurrentTime()

```
/**
   * Method that returns current time clock unit in the stream.
   *
   * @return Long: current time in stream
   **/

  def getCurrentTime: Long = {
    this.time
  }
```

B.1.7. getTotalPoints()

```
/**
   * Method that returns the total number of points processed so far in
   * the stream.
   *
   * @return Long: total number of points processed
   **/

  def getTotalPoints: Long = {
    this.N
```

```
  }
```

B.1.8. setRecursiveOutliersRMSDCheck()

```
/**
   * Method that sets if the newly created microclusters due to
   * outliers are able to absorb other outlier points. This is done
     recursively
   * for all new microclusters, thus disabling these increases slightly
     the
   * speed of the algorithm but also allows to create overlaping
     microclusters
   * at this stage.
   *
   * @param ans : true or false
   * @return Class: current class
   **/

  def setRecursiveOutliersRMSDCheck(ans: Boolean): this.type = {
    this.recursiveOutliersRMSDCheck = ans
    this
  }
```

B.1.9. setInitNormalKMeans()

```
/**
   * Changes the K-Means method to use from StreamingKmeans to
   * normal K-Means for the initialization. StreamingKMeans is much
   * faster but in some cases normal K-Means could deliver more
   * accurate initialization.
   *
   * @param ans : true or false
   * @return Class: current class
   **/
```

```
  def setInitNormalKMeans(ans: Boolean): this.type = {
    this.useNormalKMeans = ans
    this
  }
```

B.1.10. setM()

```
/**
   * Method that sets the m last number of points in a microcluster
   * used to approximate its timestamp (recency value).
   *
   * @param m : m last points
   * @return Class: current class
   **/

  def setM(m: Int): this.type = {
    this.mLastPoints = m
    this
  }
```

B.1.11. setDelta()

```
/**
   * Method that sets the threshold d, used to determine whether a
   * microcluster is safe to delete or not (Tc - d < recency).
   *
   * @param d : threshold
   * @return Class: current class
   **/

  def setDelta(d: Int): this.type = {
    this.delta = d
    this
  }
```

B.1.12. setTFactor()

```
/**
  * Method that sets the factor t of RMSDs. A point whose distance to
  * its nearest microcluster is greater than t*RMSD is considered an
  * outlier.
  *
  * @param t : t factor
  * @return Class: current class
  **/

def setTFactor(t: Double): this.type = {
  this.tFactor = t
  this
}
```

B.1.13. distanceNearestMC()

```
/**
  * Computes the distance of a point to its nearest microcluster.
  *
  * @param vec : the point
  * @param mcs : Array of microcluster information
  * @return Double: the distance
  **/

private def distanceNearestMC(vec: breeze.linalg.Vector[Double], mcs:
    Array[(MicroClusterInfo, Int)]): Double = {

  var minDist = Double.PositiveInfinity
  var i = 0
  for (mc <- mcs) {
    val dist = squaredDistance(vec, mc._1.centroid)
    if (dist != 0.0 && dist < minDist) minDist = dist
    i += 1
  }
```

```
    scala.math.sqrt(minDist)
  }
```

B.1.14. squaredDistTwoMCArrIdx()

```
  /**
    * Computes the squared distance of two microclusters.
    *
    * @param idx1 : local index of one microcluster in the array
    * @param idx2 : local index of another microcluster in the array
    * @return Double: the squared distance
    **/

  private def squaredDistTwoMCArrIdx(idx1: Int, idx2: Int): Double = {
    squaredDistance(microClusters(idx1).getCf1x :/ microClusters(idx1).
        getN.toDouble, microClusters(idx2).getCf1x :/ microClusters(idx2
        ).getN.toDouble)
  }
```

B.1.15. squaredDistPointToMCArrIdx()

```
  /**
    * Computes the squared distance of one microcluster to a point.
    *
    * @param idx1  : local index of the microcluster in the array
    * @param point : the point
    * @return Double: the squared distance
    **/

  private def squaredDistPointToMCArrIdx(idx1: Int, point: Vector[Double
      ]): Double = {
    squaredDistance(microClusters(idx1).getCf1x :/ microClusters(idx1).
        getN.toDouble, point)
  }
```

B.1.16. getArrIdxMC()

```
/**
   * Returns the local index of a microcluster for a given ID
   *
   * @param idx0 : ID of the microcluster
   * @return Int: local index of the microcluster
   **/

  private def getArrIdxMC(idx0: Int): Int = {
    var id = -1
    var i = 0
    for (mc <- microClusters) {
      if (mc.getIds(0) == idx0) id = i
      i += 1
    }
    id
  }
```

B.1.17. mergeMicroClusters()

```
  /**
    * Merges two microclusters adding all its features.
    *
    * @param idx1 : local index of one microcluster in the array
    * @param idx2 : local index of one microcluster in the array
    *
    **/

  private def mergeMicroClusters(idx1: Int, idx2: Int): Unit = {

    microClusters(idx1).setCf1x(microClusters(idx1).getCf1x :+
        microClusters(idx2).getCf1x)
    microClusters(idx1).setCf2x(microClusters(idx1).getCf2x :+
        microClusters(idx2).getCf2x)
    microClusters(idx1).setCf1t(microClusters(idx1).getCf1t +
        microClusters(idx2).getCf1t)
```

```
    microClusters(idx1).setCf2t(microClusters(idx1).getCf2t +
        microClusters(idx2).getCf2t)
    microClusters(idx1).setN(microClusters(idx1).getN + microClusters(
        idx2).getN)
    microClusters(idx1).setIds(microClusters(idx1).getIds ++
        microClusters(idx2).getIds)

    mcInfo(idx1)._1.setCentroid(microClusters(idx1).getCf1x :/
        microClusters(idx1).getN.toDouble)
    mcInfo(idx1)._1.setN(microClusters(idx1).getN)
    mcInfo(idx1)._1.setRmsd(scala.math.sqrt(sum(microClusters(idx1).cf2x
        ) / microClusters(idx1).n.toDouble - sum(microClusters(idx1).
        cf1x.map(a => a * a)) / (microClusters(idx1).n * microClusters(
        idx1).n.toDouble)))

  }
```

B.1.18. addPointMicroClusters()

```
  /**
     * Adds one point to a microcluster adding all its features.
     *
     * @param idx1  : local index of the microcluster in the array
     * @param point : the point
     *
     **/

    private def addPointMicroClusters(idx1: Int, point: Vector[Double]):
       Unit = {

    microClusters(idx1).setCf1x(microClusters(idx1).getCf1x :+ point)
    microClusters(idx1).setCf2x(microClusters(idx1).getCf2x :+ (point :*
        point))
    microClusters(idx1).setCf1t(microClusters(idx1).getCf1t + this.time)
    microClusters(idx1).setCf2t(microClusters(idx1).getCf2t + (this.time
        * this.time))
```

```
    microClusters(idx1).setN(microClusters(idx1).getN + 1)

    mcInfo(idx1)._1.setCentroid(microClusters(idx1).getCf1x :/
        microClusters(idx1).getN.toDouble)
    mcInfo(idx1)._1.setN(microClusters(idx1).getN)
    mcInfo(idx1)._1.setRmsd(scala.math.sqrt(sum(microClusters(idx1).cf2x
        ) / microClusters(idx1).n.toDouble - sum(microClusters(idx1).
        cf1x.map(a => a * a)) / (microClusters(idx1).n * microClusters(
        idx1).n.toDouble)))

  }
```

B.1.19. replaceMicroCluster()

```
  /**
     * Deletes one microcluster and replaces it locally with a new point.
     *
     * @param idx   : local index of the microcluster in the array
     * @param point : the point
     *
     **/

    private def replaceMicroCluster(idx: Int, point: Vector[Double]): Unit
        = {
      microClusters(idx) = new MicroCluster(point :* point, point, this.
          time * this.time, this.time, 1L)
      mcInfo(idx)._1.setCentroid(point)
      mcInfo(idx)._1.setN(1L)
      mcInfo(idx)._1.setRmsd(distanceNearestMC(mcInfo(idx)._1.centroid,
          mcInfo))
    }
```

B.1.20. assignToMicroCluster() - local mcInfo

```
  /**
```

```
    * Finds the nearest microcluster for all entries of an RDD.
    *
    * @param rdd     : RDD with points
    * @param mcInfo : Array containing microclusters information
    * @return RDD[(Int, Vector[Double])]: RDD that contains a tuple of
        the ID of the
    *          nearest microcluster and the point itself.
    *
    **/

  private def assignToMicroCluster(rdd: RDD[Vector[Double]], mcInfo:
      Array[(MicroClusterInfo, Int)]): RDD[(Int, Vector[Double])] = {
    rdd.map { a =>
      var minDist = Double.PositiveInfinity
      var minIndex = Int.MaxValue
      var i = 0
      for (mc <- mcInfo) {
        val dist = squaredDistance(a, mc._1.centroid)
        if (dist < minDist) {
          minDist = dist
          minIndex = mc._2
        }
        i += 1
      }
      (minIndex, a)
    }
  }
```

B.1.21. assignToMicroCluster() - broadcasted mcInfo

```
  /**
    * Finds the nearest microcluster for all entries of an RDD, uses
        broadcast variable.
    *
    * @param rdd     : RDD with points
```

```
    * @return RDD[(Int, Vector[Double])]: RDD that contains a tuple of
       the ID of the
    *         nearest microcluster and the point itself.
    *
    **/
  private def assignToMicroCluster(rdd: RDD[Vector[Double]]) = {
    rdd.map { a =>
      var minDist = Double.PositiveInfinity
      var minIndex = Int.MaxValue
      var i = 0
      for (mc <- broadcastMCInfo.value) {
        val dist = squaredDistance(a, mc._1.centroid)
        if (dist < minDist) {
          minDist = dist
          minIndex = mc._2
        }
        i += 1
      }
      (minIndex, a)
    }
  }
```

B.1.22. updateMicroClusters()

```
  /**
    * Performs all the operations to maintain the microclusters. Assign
       points that
    * belong to a microclusters, detects outliers and deals with them.
    *
    * @param assignations : RDD that contains a tuple of the ID of the
    *                       nearest microcluster and the point itself.
    *
    **/

  private def updateMicroClusters(assignations: RDD[(Int, Vector[Double
      ])]): Unit = {
```

```
var dataInAndOut: RDD[(Int, (Int, Vector[Double]))] = null
var dataIn: RDD[(Int, Vector[Double])] = null
var dataOut: RDD[(Int, Vector[Double])] = null

// Calculate RMSD
if (initialized) {
  dataInAndOut = assignations.map { a =>
    val nearMCInfo = broadcastMCInfo.value.find(id => id._2 == a._1)
        .get._1
    val nearDistance = scala.math.sqrt(squaredDistance(a._2,
        nearMCInfo.centroid))

    if (nearDistance <= tFactor * nearMCInfo.rmsd) (1, a)
    else (0, a)
  }
}

// Separate data
if (dataInAndOut != null) {
  dataIn = dataInAndOut.filter(_._1 == 1).map(a => a._2)
  dataOut = dataInAndOut.filter(_._1 == 0).map(a => a._2)
} else dataIn = assignations

// Compute sums, sums of squares and count points... all by key
logInfo(s"Processing points")

// sumsAndSumsSquares -> (key: Int, (sum: Vector[Double], sumSquares
    : Vector[Double], count: Long ) )
val sumsAndSumsSquares = timer {
  val aggregateFuntion = (aa: (Vector[Double], Vector[Double], Long)
      , bb: (Vector[Double], Vector[Double], Long)) => (aa._1 :+ bb.
      _1, aa._2 :+ bb._2, aa._3 + bb._3)
  dataIn.mapValues(a => (a, a :* a, 1L)).reduceByKey(
      aggregateFuntion).collect()
}
```

```
var totalIn = 0L

for (mc <- microClusters) {
  for (ss <- sumsAndSumsSquares) if (mc.getIds(0) == ss._1) {
    mc.setCf1x(mc.cf1x :+ ss._2._1)
    mc.setCf2x(mc.cf2x :+ ss._2._2)
    mc.setN(mc.n + ss._2._3)
    mc.setCf1t(mc.cf1t + ss._2._3 * this.time)
    mc.setCf2t(mc.cf2t + ss._2._3 * (this.time * this.time))
    totalIn += ss._2._3
  }
}



logInfo(s"Processing " + (currentN - totalIn) + " outliers")
timer {
  if (dataOut != null && currentN - totalIn != 0) {
    var mTimeStamp: Double = 0.0
    val recencyThreshold = this.time - delta
    var safeDeleteMC: Array[Int] = Array()
    var keepOrMergeMC: Array[Int] = Array()
    var i = 0


    for (mc <- microClusters) {
      val meanTimeStamp = if (mc.getN > 0) mc.getCf1t.toDouble / mc.
          getN.toDouble else 0
      val sdTimeStamp = scala.math.sqrt(mc.getCf2t.toDouble / mc.
          getN.toDouble - meanTimeStamp * meanTimeStamp)

      if (mc.getN < 2 * mLastPoints) mTimeStamp = meanTimeStamp
      else mTimeStamp = Gaussian(meanTimeStamp, sdTimeStamp).icdf(1
          - mLastPoints / (2 * mc.getN.toDouble))
```

```
    if (mTimeStamp < recencyThreshold || mc.getN == 0)
      safeDeleteMC = safeDeleteMC :+ i
    else keepOrMergeMC = keepOrMergeMC :+ i

    i += 1
  }

  var j = 0
  var newMC: Array[Int] = Array()


  for (point <- dataOut.collect()) {

    var minDist = Double.PositiveInfinity
    var idMinDist = 0
    if (recursiveOutliersRMSDCheck) for (id <- newMC) {
      val dist = squaredDistPointToMCArrIdx(id, point._2)
      if (dist < minDist) {
        minDist = dist
        idMinDist = id
      }

    }

    if (scala.math.sqrt(minDist) <= tFactor * mcInfo(idMinDist)._1
      .rmsd) addPointMicroClusters(idMinDist, point._2)
    else if (safeDeleteMC.lift(j).isDefined) {
      replaceMicroCluster(safeDeleteMC(j), point._2)
      newMC = newMC :+ safeDeleteMC(j)
      j += 1
    } else {
      var minDist = Double.PositiveInfinity
      var idx1 = 0
      var idx2 = 0

      for (a <- keepOrMergeMC.indices)
        for (b <- (0 + a) until keepOrMergeMC.length) {
```

```
                var dist = Double.PositiveInfinity
                if (keepOrMergeMC(a) != keepOrMergeMC(b)) dist =
                    squaredDistance(mcInfo(keepOrMergeMC(a))._1.centroid
                    , mcInfo(keepOrMergeMC(b))._1.centroid)
                if (dist < minDist) {
                  minDist = dist
                  idx1 = keepOrMergeMC(a)
                  idx2 = keepOrMergeMC(b)
                }
              }
            mergeMicroClusters(idx1, idx2)
            replaceMicroCluster(idx2, point._2)
            newMC = newMC :+ idx2
          }
        }
      }
    }
  }
```

B.2. CluStream class

```
package com.backhoff.clustream

import breeze.linalg._
import org.apache.spark.{SparkContext, Logging}
import org.apache.spark.streaming.dstream.DStream
import org.apache.spark.annotation.Experimental
import java.io._
import java.nio.file.{Paths, Files}
import org.apache.spark.mllib.clustering.KMeans

/**
  * Class that contains the offline methods for the CluStream
  * method. It can be initialized with a CluStreamOnline model to
  * facilitate the use of it at the same time the online process
  * is running.
```

```
  *
  **/

@Experimental
class CluStream (
                    val model:CluStreamOnline)
  extends Logging with Serializable{

  def this() = this(null)

  private def sample[A](dist: Map[A, Double]): A = {...}

  def saveSnapShotsToDisk(dir: String = "", tc: Long, alpha: Int = 2, l:
      Int = 2): Unit = {...}

  def getSnapShots(dir: String = "", tc: Long, h: Long): (Long,Long) =
     {...}

  def getMCsFromSnapshots(dir: String = "", tc: Long, h: Long): Array[
     MicroCluster] = {...}

  def getCentersFromMC(mcs: Array[MicroCluster]): Array[Vector[Double]]
     = {...}

  def getWeightsFromMC(mcs: Array[MicroCluster]): Array[Double] = {...}

  def fakeKMeans(sc: SparkContext,k: Int, numPoints: Int, mcs: Array[
     MicroCluster]): org.apache.spark.mllib.clustering.KMeansModel =
     {...}

  def startOnline(data: DStream[breeze.linalg.Vector[Double]]): Unit =
     {...}
}
```

B.2.1. sample()

```
/**
  * Method that samples values from a given distribution.
  *
  * @param dist: this is a map containing values and their weights in
  *              the distributions. Weights must add to 1.
  *              Example. {A -> 0.5, B -> 0.3, C -> 0.2 }
  * @return A: sample value A
  **/

private def sample[A](dist: Map[A, Double]): A = {
  val p = scala.util.Random.nextDouble
  val it = dist.iterator
  var accum = 0.0
  while (it.hasNext) {
    val (item, itemProb) = it.next
    accum += itemProb
    if (accum >= p)
      return item
  }
  sys.error(f"this should never happen") // needed so it will compile
}
```

B.2.2. saveSnapshotsToDisk()

```
/**
   * Method that saves a snapshot to disk using the pyramidal time
   * scheme to a given directory.
   *
   * @param dir: directory to save the snapshot

   * @param tc: time clock unit to save
   * @param alpha: alpha parameter of the pyramidal time scheme
   * @param l: l modifier of the pyramidal time scheme
   **/
```

```
def saveSnapShotsToDisk(dir: String = "", tc: Long, alpha: Int = 2, l:
    Int = 2): Unit ={

  var write = false
  var delete = false
  var order = 0
  val mcs = model.getMicroClusters


  val exp = (scala.math.log(tc) / scala.math.log(alpha)).toInt

    for (i <- 0 to exp) {
      if (tc % scala.math.pow(alpha, i + 1) != 0 && tc % scala.math.
          pow(alpha, i) == 0) {
        order = i
        write = true
      }
    }

    val tcBye = tc - ((scala.math.pow(alpha, l) + 1) * scala.math.pow(
        alpha, order + 1)).toInt

    if (tcBye > 0) delete = true

    if (write) {
      val out = new ObjectOutputStream(new FileOutputStream(dir + "/"
          + tc))

      try {
        out.writeObject(mcs)
      }
      catch {
        case ex: IOException => println("Exception while writing file "
            + ex)
      }
      finally {
        out.close()
```

```
        }
      }

      if (delete) {
        try {
          new File(dir + "/" + tcBye).delete()
        }
        catch {
          case ex: IOException => println("Exception while deleting file
              " + ex);
        }
      }
  }
```

B.2.3. getSnapshots()

```
/**
   * Method that gets the snapshots to use for a given time and horizon
       in a
   * given file directory.
   *
   * @param dir: directory to save the snapshot

   * @param tc: time clock unit to save
   * @param h: time horizon
   * @return (Long,Long): tuple of the first and second snapshots to
      use.
   **/

  def getSnapShots(dir: String = "", tc: Long, h: Long): (Long,Long) = {

    var tcReal = tc
    while(!Files.exists(Paths.get(dir + "/" + tcReal)) && tcReal >= 0)
       tcReal = tcReal - 1
    var tcH = tcReal - h
```

```
    while(!Files.exists(Paths.get(dir + "/" + tcH)) && tcH >= 0) tcH =
        tcH - 1
    if(tcH < 0) while(!Files.exists(Paths.get(dir + "/" + tcH))) tcH =
        tcH + 1

    if(tcReal == -1L) tcH = -1L
    (tcReal, tcH)
  }
```

B.2.4. getMCsFromSnapshots()

```
/**
   * Method that returns the microclusters from the snapshots for a
     given time and horizon in a
   * given file directory. Subtracts the features of the first one with
     the second one.
   *
   * @param dir: directory to save the snapshot

   * @param tc: time clock unit to save
   * @param h: time horizon
   * @return Array[MicroCluster]: computed array of microclusters
   **/

  def getMCsFromSnapshots(dir: String = "", tc: Long, h: Long): Array[
      MicroCluster] = {
    val (t1,t2) = getSnapShots(dir,tc,h)

    try{
      val in1 = new ObjectInputStream(new FileInputStream(dir + "/" + t1
          ))
      val snap1 = in1.readObject().asInstanceOf[Array[MicroCluster]]

      val in2 = new ObjectInputStream(new FileInputStream(dir + "/" + t2
          ))
      val snap2 = in2.readObject().asInstanceOf[Array[MicroCluster]]
```

```
      in2.close()
      in1.close()

      val arrs1 = snap1.map(_.getIds)
      val arrs2 = snap2.map(_.getIds)

      val relatingMCs = snap1 zip arrs1.map(a => arrs2.zipWithIndex.map(
          b=> if(b._1.toSet.intersect(a.toSet).nonEmpty) b._2;else -1))
      relatingMCs.map{ mc =>
        if (!mc._2.forall(_ == -1) && t1 - h >= t2) {
          for(id <- mc._2) if(id != -1) {
            mc._1.setCf2x(mc._1.getCf2x :- snap2(id).getCf2x)
            mc._1.setCf1x(mc._1.getCf1x :- snap2(id).getCf1x)
            mc._1.setCf2t(mc._1.getCf2t - snap2(id).getCf2t)
            mc._1.setCf1t(mc._1.getCf1t - snap2(id).getCf1t)
            mc._1.setN(mc._1.getN - snap2(id).getN)
            mc._1.setIds(mc._1.getIds.toSet.diff(snap2(id).getIds.toSet)
                .toArray)
          }
          mc._1
        }else mc._1

      }
    }
    catch{
      case ex: IOException => println("Exception while reading files " +
          ex)
      null
    }

  }
```

B.2.5. getCentersFromMC()

```
/**
```

```
    * Method that returns the centrois of the microclusters.
    *
    * @param mcs: array of microclusters
    * @return Array[Vector]: computed array of centroids
    **/

  def getCentersFromMC(mcs: Array[MicroCluster]): Array[Vector[Double]]
     = {
    mcs.filter(_.getN > 0).map(mc => mc.getCf1x :/ mc.getN.toDouble)
    }
```

B.2.6. getWeightsFromMC()

```
/**
    * Method that returns the weights of the microclusters from the
       number of points.
    *
    * @param mcs: array of microclusters
    * @return Array[Double]: computed array of weights
    **/

  def getWeightsFromMC(mcs: Array[MicroCluster]): Array[Double] = {
    var arr: Array[Double] = mcs.map(_.getN.toDouble).filter(_ > 0)
    val sum: Double = arr.sum
    arr.map(value => value/sum)
  }
```

B.2.7. fakeKMeans()

```
/**
    * Method that returns a computed KMeansModel. It runs a modified
       version
    * of the KMeans algorithm in Spark from sampling the microclusters
       given
    * its weights.
```

```
    *
    * @param sc: spark context where KMeans will run
    * @param k: number of clusters
    * @param mcs: array of microclusters
    * @return org.apache.spark.mllib.clustering.KMeansModel: computed
      KMeansModel
    **/

  def fakeKMeans(sc: SparkContext,k: Int, numPoints: Int, mcs: Array[
      MicroCluster]): org.apache.spark.mllib.clustering.KMeansModel ={

      val kmeans = new KMeans()
      var centers = getCentersFromMC(mcs).map(v => org.apache.spark.
          mllib.linalg.Vectors.dense(v.toArray))
      val weights = getWeightsFromMC(mcs)
      val map = (centers zip weights).toMap
      val points = Array.fill(numPoints)(sample(map))


      kmeans.setMaxIterations(20)
      kmeans.setK(k)
      kmeans.setInitialModel(new org.apache.spark.mllib.clustering.
          KMeansModel(Array.fill(k)(sample(map))))
      val trainingSet = sc.parallelize(points)
      val clusters = kmeans.run(trainingSet)
      trainingSet.unpersist(blocking = false)
      clusters

  }
```

B.2.8. startOnline()

```
  /**
     * Method that allows to run the online process from this class.
     *
     * @param data: data that comes from the stream
```

```
    *
    **/

  def startOnline(data: DStream[breeze.linalg.Vector[Double]]): Unit ={
    model.run(data)
  }
```

B.3. MicroCluster class

```
 /**
  * Packs the microcluster object and its features in one single class
  *
  **/

protected class MicroCluster(
                              var cf2x: breeze.linalg.Vector[Double],
                              var cf1x: breeze.linalg.Vector[Double],
                              var cf2t: Long,
                              var cf1t: Long,
                              var n: Long,
                              var ids: Array[Int]) extends Serializable
                                 {

  def this(cf2x: breeze.linalg.Vector[Double], cf1x: breeze.linalg.
      Vector[Double], cf2t: Long, cf1t: Long, n: Long) = this(cf2x, cf1x
      , cf2t, cf1t, n, Array(MicroCluster.inc))

  def setCf2x(cf2x: breeze.linalg.Vector[Double]): Unit = {
    this.cf2x = cf2x
  }

  def getCf2x: breeze.linalg.Vector[Double] = {
    this.cf2x
  }

  def setCf1x(cf1x: breeze.linalg.Vector[Double]): Unit = {
```

```
    this.cf1x = cf1x
  }

  def getCf1x: breeze.linalg.Vector[Double] = {
    this.cf1x
  }

  def setCf2t(cf2t: Long): Unit = {
    this.cf2t = cf2t
  }

  def getCf2t: Long = {
    this.cf2t
  }

  def setCf1t(cf1t: Long): Unit = {
    this.cf1t = cf1t
  }

  def getCf1t: Long = {
    this.cf1t
  }

  def setN(n: Long): Unit = {
    this.n = n
  }

  def getN: Long = {
    this.n
  }

  def setIds(ids: Array[Int]): Unit = {
    this.ids = ids
  }

  def getIds: Array[Int] = {
    this.ids
```

```
  }
}
```

B.4. MicroClusterInfo class

```
/**
 * Packs some microcluster information to reduce the amount of data to
    be
 * broadcasted.
 *
 **/

private class MicroClusterInfo(
                                var centroid: breeze.linalg.Vector[
                                    Double],
                                var rmsd: Double,
                                var n: Long) extends Serializable {

  def setCentroid(centroid: Vector[Double]): Unit = {
    this.centroid = centroid
  }

  def setRmsd(rmsd: Double): Unit = {
    this.rmsd = rmsd
  }

  def setN(n: Long): Unit = {
    this.n = n
  }
}
```

Bibliography

[1] Apache spark. http://spark.apache.org/. Accessed: 08.07.2015.

[2] Apache spark: Preparing for the next wave of reactive big data. https://www.typesafe.com/blog/apache-spark-preparing-for-the-next-wave-of-reactive-big-data. January 27, 2015.

[3] Apache spark, research. http://spark.apache.org/research. Accessed: 24.01.2016.

[4] Apache spark, streaming. http://spark.apache.org/docs/latest/streaming-programming-guide.html. Accessed: 01.03.2016.

[5] Databricks, streaming k-means. https://databricks.com/blog/2015/01/28/introducing-streaming-k-means-in-spark-1-2.html. Accessed: 01.03.2016.

[6] Scikit-learn: Mini batch kmeans. http://scikit-learn.org/stable/modules/clustering.html. 2010-2014 BSD License, Accessed: 08.07.2015.

[7] Spark's mlib: Clustering. http://spark.apache.org/docs/latest/mllib-clustering.html. Accessed: 08.07.2015.

[8] Charu C. Aggarwal, Jiawei Han, Jianyong Wang, and Philip S. Yu. A framework for clustering evolving data streams. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, pages 81–92. VLDB Endowment, 2003.

[9] E. Alpaydin. *Introduction to Machine Learning*. Adaptive Computation and Machine Learning. MIT Press, 2014.

[10] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[11] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI 04: PROCEEDINGS OF THE 6TH CONFERENCE ON SYMPOSIUM ON OPERATING SYSTEMS DESIGN AND IMPLEMENTATION*. USENIX Association, 2004.

[12] D. Garg and K. Trivedi. Fuzzy k-mean clustering in mapreduce on cloud based hadoop. In *Advanced Communication Control and Computing Technologies (ICACCCT), 2014 International Conference on*, pages 1607–1610, May 2014.

[13] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[14] Satish Gopalani and Rohan Arora. Article: Comparing apache spark and map reduce with performance analysis using k-means. *International Journal of Computer Applications*, 113(1):8–11, March 2015. Full text available.

[15] Hai-Guang Li, Gong-Qing Wu, Xue-Gang Hu, Jing Zhang, Lian Li, and Xindong Wu. K-means clustering with bagging and mapreduce. In *System Sciences (HICSS), 2011 44th Hawaii International Conference on*, pages 1–8, Jan 2011.

[16] SimoneA. Ludwig. Mapreduce-based fuzzy c-means clustering algorithm: implementation and scalability. *International Journal of Machine Learning and Cybernetics*, pages 1–12, 2015.

[17] Gianmarco De Francisci Morales and Albert Bifet. Samoa: Scalable advanced massive online analysis. *Journal of Machine Learning Research*, 16:149–153, 2015.

[18] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. BIRCH: an efficient data clustering method for very large databases. In Jennifer Widom, editor, *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 103–114. ACM Press, 1996.