Scalable Online-Offline Stream Clustering in Apache Spark

Omar Backhoff
Technische Universität München (TUM)
Munich, Germany
omarbackhoff@gmail.com

Eirini Ntoutsi
Leibniz Universität Hannover
Hanover, Germany
ntoutsi@kbs.uni-hannover.de

Abstract—Two of the most popular approaches for dealing with big data are distributed computing and stream mining. In this paper, we incorporate both approaches in order to bring a competitive stream clustering algorithm, namely CluStream [8], into a modern framework for distributed computing, namely, Apache Spark. CluStream is one of the most popular clustering approaches for stream clustering and the one that introduced the online-offline mining process: the online phase summarizes the stream through statistical summaries and the offline phase generates the final clusters upon these summaries. We obtain a scalable stream clustering method which is open source and can be used by the Apache Spark community. Our experiments show that our adaptation, Spark-CluStream, achieves similar quality to the original approach [8], while it is more efficient.

Keywords—big data streams; stream mining; stream clustering; CluStream; Apache Spark

I. INTRODUCTION

The analysis of data streams comes with important questions: what kind of data is it? What important information is contained in the stream? How does the stream evolve? The later comprises the key question for our work, i.e., dealing with the evolution of the stream. We focus on the clustering task, i.e., how can we maintain a valid clustering structure in a volatile and evolving data stream environment.

Clustering is one of the main tasks in data mining, also referred as an exploratory task as it allows us to get to know our data. The goal is to find groups of objects that share similar characteristics. One of the most popular algorithms for stream clustering is CluStream [8], proposed in 2003, which provides more information than previously developed algorithms for data stream clustering by that time.

Recently, distributed computing frameworks are employed to deal with big data challenges and many algorithms have been adapted to these settings. In this work, we adapt CluStream to the Apache Spark framework in order to obtain a distributed stream clustering algorithm that can deal with clustering over big data streams. As CluStream was not designed for a distributed setting, the adaptation is not straightforward and involves re-designing core parts of the algorithm.

II. RELATED WORK / BASIC CONCEPTS

A. CluStream

CluStream is a method developed in the Watson Research Center at IBM and the University of Illinois, UIUC. This method presented a different approach on the matter of clustering streams of data with respect to a modified version of K-Means which was adapted to work also with data streams. The main difference relies on the separation of 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 process the results of the former to produce the actual clusters wanted (offline part).

Separating the clustering process provides the user 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 in 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.

1) 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: that is the given name for the statistical information summaries that is computed during the online component. They are a temporal extension of cluster feature vectors [18], which benefit from an additive feature that makes them a natural choice for the data stream problem [8].

- Pyramidal time frame: micro-clusters are stored periodically following a pyramidal pattern. This allows a nice tradeoff between the ability to store large amounts of information while giving the user the possible 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.

2) Maintaining the micro-clusters: Whenever a new point arrives, it is necessary to find its nearest micro-cluster. It is possible to calculate an average radius or RMSD, only to then compare the distance to the point to a factor of it: when the distance between a point and its nearest micro-cluster is smaller or equal to the average radius (of the micro-cluster in question) times a user defined factor, then this point is added to the micro-cluster. Adding a point to a micro-cluster means that the properties of the micro-cluster change, such as RMSD and size (number of points).

Whenever a point (outlier) does not fulfill the mentioned condition, then a new micro-cluster has to be created in order to give this point a chance as a potential new cluster. In order to do so, an older micro-cluster has to be deleted or two micro-clusters have to be merged. To determine which solution is appropriate, a recency value for each micro-cluster has to be determined1 and until all the micro-clusters which have an older recency value than a user specified parameter are deleted, it is possible to start merging the micro-clusters which are closest to one another.

3) Offline macro-clustering: The macro-clustering part is done by selecting a time window and then performing a modified version of K-Means to cluster the center of the current micro-clusters using the size as weights.

B. SPARK

Apache Spark is an open source framework developed in the AMPLab at the University of California. Traditionally, MapReduce and DAG engines are based on an acyclic data flow, which makes them non 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.

The solution proposed offers 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.

C. SPARK streaming

Figure 1 shows the general idea of Spark streaming [4], a raw stream is linked to this module which gets transformed into 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 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.

D. Existing CluStream implementations in distributed computing frameworks

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.

StreamDM is a collection of algorithms for mining big data streams 2. 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.

III. CluSTREAM IN APACHE SPARK

There are some modifications which had to be done in order to adapt CluStream in Spark. Working with Spark means working distributed computing and, thus, the algorithm has to be able to work in parallel. Both parts (online and offline) were adapted.

1See [8] for more details.

2As it is stated by them here: http://huawei-noah.github.io/streamDM/
A. CluStreamOnline class (online phase)

Two processes were modified: processing the stream and updating the micro-clusters. As this adaptation uses Spark Streaming, the points coming from the stream are processed in batches at user specified time intervals. This contrasts with the original methodology which indicates to process point by point.

1) Finding nearest micro-cluster: The maintenance of the micro-clusters starts with this operation. After initialization (described in [8]) is performed, finding the nearest micro-clusters starts with this operation. After initialization point by point.

Algorithm 1 Find nearest micro-cluster.

```
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.centroid)
6:     if distance ≤ minDistance then
7:       minDistance ← distance
8:       minIndex ← mc mcID
9:   end if
10: end for
11: p = (minIndex, p)
12: end for return rdd
```

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 1 describes a simple search for the minimum distance for every point in the RDD to the micro-clusters.

At this point, every node performs this operation to find the nearest micro-cluster for all the points it locally have.

2) Processing points: The points are separated in two: points within micro-clusters and outliers, 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 in a batch is treated equally in terms of temporal properties as this batch gets distributed among the cluster with the same time stamp and there is no constant communication among nodes.
- The process of handling outliers involves deleting and merging micro-clusters for every outlier, modifying the micro-clusters’ structure. A sequential implementation was preferred to reduce communication costs.

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.

3) Handling outliers: First the micro-clusters which are safe to delete are determined, then the outliers can be handled. In general, there are three possible scenarios for outliers:

- If the point lies within the restriction regarding the RMSD for its nearest micro-cluster in the array `newMicroClusters`, the point is added to it. This stores all newly created micro-clusters.
- If the point does not lie within any of the new micro-clusters, then it replaces a micro-cluster from the `safeDelete` array, while there are safe-to-delete micro-clusters.
- If none of the previous scenarios are viable, then the two micro-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 $O(pm d \cdot \frac{q!}{(q-m)!})$, 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.

Algorithm 2 handle outliers.

```
1: j ← 0
2: for all p ∈ dataOut do
3:   distance, mcID ← getMinDistanceFromIDs(newMicroClusters, p2)
4:   if distance < t = mcInfo(mcID).rmsd then
5:     addPointToMicroCluster(mcID, p2)
6:   else
7:     if safeDelete[j].isDefined then
8:       replaceMicroCluster(safeDelete[j], p2)
9:       newMicroClusters.append(j)
10:     j ← j + 1
11:   end if
12:   index1, index2 ← getTwoClosestMicroClusters(keepOrMerge)
13:   mergeMicroClusters(index1, index2)
14:   replaceMicroClusters(index2, p2)
15:   newMicroClusters.append(j)
16:   j ← j + 1
17: end if
18: end if
19: end for
```
It is important in the procedure described in Algorithm 2 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.

B. CluStream class (offline phase)

Using a weighted K-Means approach, as described in [8] was not directly possible, and for that reason, a new adaptation had to be done in order to achieve similar results.

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 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.

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.

IV. EXPERIMENTS

A. Experiments setting

For the experiments we used the Network Intrusion dataset\(^3\), which consists of 494,021 instances. For the analysis, we used only the numerical attributes (#34 out of #43 attributes). We vary the speed of the stream and the horizon and we derive two different stream configurations. The first one, denoted as DS1, has a speed \(v = 2,000\) points per timestamp, which implies that it lasts for \(\frac{494,021}{2,000}\) \(\approx 247\) time units, and a horizon \(H = 1\). The second one, denoted as DS2, has a speed of \(v = 200\) points per timestamp, thus lasting for a period of \(\frac{494,021}{200}\) \(\approx 2470\) time points, and a horizon \(H = 256\). To evaluate the clustering quality, we report in Section IV-B on the sum of square distances (SSQ) from the points to their nearest micro-cluster, using Euclidean distance as the distance function, within a horizon \(H\). Efficiency and scalability is discussed in Section IV-C.

B. Clustering quality

We first compare Spark-CluStream to the original CluStream (Section IV-B1) and then against other stream clustering approaches in Spark (Section IV-B2).

1) Quality of Spark – CluStream vs original CluStream:

Results for DS1: We used the same parameters as in [8], i.e., \(\alpha = 2, l = 10, \text{InitNumber} = 2000, \delta = 512, t = 2\). The parameter \(m\), for \(m\) last points, was the only one not provided, we set it to \(m = 20\). \(m\) is used to determine the approximate recency value as if the time of arrival of the last \(m\) points was averaged. For DS1, both \(m\) and \(\delta\) are irrelevant and the reason is that the threshold is never reached (247 time units vs. 512). The number of micro-clusters was set to \(q = 50\), 10 times the number of final clusters (5).

In Table I we show the SSQ scores for our Spark-CluStream and the results from CluStream as reported in [8] - these are approximate results “extracted” from the original paper. Although we don’t know the exact values for CluStream, we can see that the magnitudes of the average SSQ are comparable.

Results for DS2: Parameters were set as for DS1. The exact scores of Spark-CluStream and the approximate values from CluStream are reported in Table II. Again the quality scores are comparable, i.e., our Spark-CluStream achieves similar quality clusterings to the original CluStream.

2) Spark–CluStream vs other clustering approaches in SPARK: We compare our Spark-CluStream against available solutions for stream clustering in Spark and in particular against Streaming K-Means \(^4\) and StreamDM-CluStream \(^5\).

We report here on their clustering quality, the efficiency issue is discussed in Section IV-C.

We roughly overview these methods hereafter.

- Streaming K-Means \(^6\):
  - An 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. As it also considers the number of points, after some time "old" data will be considerably larger than new batches.

- StreamDM-CluStream \(^7\):
  - This adaptation of CluStream does not include the offline part as a separate module, meaning that

\(^3\)Source: http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html


\(^5\)More information: http://huawei-noah.github.io/streamDM/


\(^7\)More information: http://huawei-noah.github.io/streamDM/
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 \( \delta \) parameter is used instead as an equivalent, relying on the micro-clustering part only and its ability to delete and create new micro-clusters.

**Results on DS1:** For the DS1 stream the results are shown in Figure 3. For *Streaming K-Means*, the horizon \( H = 1 \) is transformed to \( \frac{1}{2} \text{Life} = 1,000 \) points, as the stream speed is 2,000 points. The \( \text{decayFactor} \) is set to 0, i.e., only the last 2,000 points will influence on the clusters. *StreamDM-CluStream* is set up with its default parameters, only changing the horizon to 1 and the number of micro-clusters to 50 to match those of *Spark-CluStream*. As we can see, *Spark-CluStream* delivers results which are very close to those of *Streaming K-Means*, whereas *Streaming K-Means* with the \( \text{decayFactor} \) (DF) is the best. The surprising results come from *StreamDM-CluStream*, which is the worse among the tested methods, especially for the last two points, i.e., at 160k and 320k. To further investigate this, we conduct another experiment where we compare *Spark-CluStream* without snapshots against *StreamDM-CluStream*. The results are shown in Figure 4.

As we can see, *Spark-CluStream* outperforms *StreamDM-CluStream* even if we remove the snapshot part, but the quality is lower comparing to the snapshot version.

**Results on DS2:** For the DS2 stream the results are shown in Figure 5. All parameters remained the same for all methods, except for the \( \frac{1}{2} \text{Life} \) parameter for *Streaming K-Means*, which is set to \( \frac{1}{2} \text{Life} = 25,600 \) (200 \( \times \) 256 = 51,200). We calculate the \( \text{decayFactor} \) as follows: At 150,000 points, the ratio of the points to cluster to the total number of points at that particular time is \( \frac{51,200}{150,000} \approx 0.3413 \). At 250,000 points, this equals to \( \frac{51,200}{250,000} \approx 0.2048 \). At 350,000 points, this equals to \( \frac{51,200}{350,000} \approx 0.1462 \). At 450,000 points, this equals to \( \frac{51,200}{450,000} \approx 0.1137 \). Averaging those ratios leads to a \( \text{decayFactor} = 0.2051 \). As we can see, *Spark-CluStream* performs consistently well and better than *StreamDM-CluStream*. As expected *Streaming K-Means* with the \( \text{decayFactor} \) achieves the best performance.

We repeat the without-snapshot experiment, the results are shown in Figure 6. As we can see, *Spark-CluStream* delivers better results than *StreamDM-CluStream* but the difference is reduced significantly comparing to DS1. These results might indicate that *StreamDM-CluStream* benefits from larger horizons.

---

**Table I:** DS1 - Average SSQ values

<table>
<thead>
<tr>
<th></th>
<th>10k</th>
<th>40k</th>
<th>160k</th>
<th>320k</th>
</tr>
</thead>
<tbody>
<tr>
<td>CluStream</td>
<td>10^4</td>
<td>10^4</td>
<td>10^4</td>
<td>10^4</td>
</tr>
<tr>
<td>Spark-CluStream</td>
<td>3.059 \times 10^4</td>
<td>6.016 \times 10^4</td>
<td>7.813 \times 10^4</td>
<td>4.191 \times 10^4</td>
</tr>
</tbody>
</table>

**Table II:** DS2 - Average SSQ values

<table>
<thead>
<tr>
<th></th>
<th>150k</th>
<th>250k</th>
<th>350k</th>
<th>450k</th>
</tr>
</thead>
<tbody>
<tr>
<td>CluStream</td>
<td>10^4</td>
<td>10^4</td>
<td>10^4</td>
<td>10^5</td>
</tr>
<tr>
<td>Spark-CluStream</td>
<td>5.402 \times 10^4</td>
<td>5.143 \times 10^4</td>
<td>1.892 \times 10^4</td>
<td>9.646 \times 10^4</td>
</tr>
</tbody>
</table>

![Figure 3: Average SSQ for different stream clustering methods in SPARK. DS1 (Stream speed v = 2,000, H = 1) (stream speed v = 2,000, H = 1) ](image)

![Figure 4: Spark-CluStream without snapshots vs StreamDM-CluStream. DS1 (Stream speed v = 2,000, H = 1, m = 100).](image)
C. Scalability

We test the scalability with respect to data dimensionality and number of microclusters, using data generated by a Random Radial Basis Function generator.

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 as the number of attributes and the number of final clusters for a specific purpose are two key factors which determine the complexity of Spark-CluStream. 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. An application using Spark streaming assigns one core exclusively to handle the stream, therefore the number of processors mentioned in here is the total, but the real number of processors used for the computations is that number minus one.

Figure 7 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 needed to process the points in every batch of data and here is where Spark-CluStream shows its scalability, which is almost linear\(^8\) for up to 16-17 processors, as it can be seen in Figure 8. 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 9.

Finally, when the number of clusters and the number of attributes are both increased significantly, Figure 10 shows for Spark-CluStream quasi-linear scalability but this time only up to about 8-9 processors. After that point, the speedup slows down showing almost no improvement after 16 processors. This test never reached a stable configuration.

D. Performance

In this section, the scalability of Spark-CluStream is compared to that of StreamDM-CluStream and Spark’s Streaming K-Means using the Spark cluster setup for \(q = 20\) and \(d = 100\), for the CluStream method. Also, a test on a single machine is performed.

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 Figure 11, it seems to scale as fast as Spark-CluStream but it was not enough even with 40 processors.

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.

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 12 are quite remarkable. As StreamDM-CluStream shows a very significant disadvantage when using greater numbers of micro-clusters and higher dimensions.

\(^8\)By linear scalability does not mean it scales with a 1 to 1 ratio, but rather linearly proportional.
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 for this matter.

V. CONCLUSIONS

We proposed a variation of CluStream tailored to Apache Spark, Spark-CluStream. Our experiments show that Spark-CluStream achieves similar quality to the original approach [8], while it is far more efficient. Comparing to other stream clustering approaches in Spark, our results show that Streaming K-Means is the fastest algorithm among the three tested (highly optimized for Spark), but it does not offer the flexibility of the online-offline clustering approaches like CluStream that better fit evolving data streams. Comparing Spark-CluStream to StreamDM-CluStream, Spark-CluStream delivers more consistent and accurate results, and outperforms StreamDM-CluStream in most cases, including one up to around 18 times faster.

As part of our future work, we will focus on minimizing the communication cost between the different nodes to
Figure 11: Processing time comparison: $q = 20$, $d = 100$

Figure 12: Processing time comparison for a single machine: $q = 50$, $d = 34$

further improve the efficiency and also on dealing with outliers to improve quality.

A. Final comments

A package Spark-CluStream exists in spark-packages.org, which can be installed to an existing SPARK installation. There, a link to the GitHub repository can be found.

REFERENCES


