Scalable Data Stream Clustering

Master Thesis

by

Biplob Biswas

Submitted to the Faculty IV, Electrical Engineering and Computer Science
Database Systems and Information Management Group
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

as part of the ERASMUS MUNDUS programme IT4BI

at the

TECHNISCHE UNIVERSITÄT BERLIN

July 31, 2016

Thesis Advisors:
Behrouz Derakhshan, M.Sc.

Thesis Supervisor:
Prof. Dr. Volker Markl

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Berlin, July 31, 2016

Biplob Biswas

Acknowledgements

I would like to thank Professor Dr. Volker Markl, for his support and supervision throughout the thesis and otherwise. I would also like to thank my project advisor Behrouz Derakshan for his advice and support during the time it took me to acquire knowledge about stream clustering algorithms, and for his honest input in everything related to the thesis, without which this thesis wouldn't have been possible.

Also, I would like to thank everyone at DIMA for providing all the inputs and resources whenever I needed them.

Scalable Data Stream Clustering

by Biplob Biswas

Database Systems and Information Management Group

Electrical Engineering and Computer Science

Master of Information Technology for Business Intelligence

Abstract

As Amazon AWS[1] describes perfectly "Streaming Data is data that is generated continuously by thousands of data sources, which typically send in the data records simultaneously, and in small sizes (order of Kilobytes)". The sudden interest in Streaming Data or Data Streams is fuelled by the growing trend in smart devices which all act as data streams. The need to analyse data from such sources are at an all time high. One of the very popular and most widely used unsupervised machine learning algorithm for analysing any data is clustering. It's main goal is to segregate and group together objects having similar properties. Clustering has widespread use such as medical diagnosis, anomaly detection, finding similar documents (like in News recommendation) and crime analysis. These applications of clustering makes it very interesting to perform analysis over aforementioned data stream sources.

Performing clustering over data streams is non-trivial. This is because of the fact that traditional algorithms uses multiple passes over the data to create clusters, whereas in data streams, mostly one pass over the data is possible(in a few scenarios more than one pass is possible). For this purpose, many stream clustering algorithms have been proposed and this thesis focuses on such algorithms and aims to present a holistic survey of such existing algorithms.

The other crucial task of this thesis is to select one of the stream clustering algorithms and implement it in a distributed and scalable environment. This again is non-trivial as majority of stream clustering algorithms are implemented and tested on a single machine environment. For implementation, Apache Flink is chosen as a platform of choice which is a scalable batch and stream processing engine. The results are then compared against results obtained using MOA, an open source data stream mining framework which only works on a single machine. The comparison results are then compiled and presented in the thesis.

Contents

Chapter 1

Introduction

Data stream mining has gained a lot of attention in recent years. It has been actively studied and different methods and algorithms have been proposed. Clustering is a common unsupervised machine learning task, whose goal is to group similar items together. It has applications in medical diagnoses, finding similar documents for example in News recommendation, crime analysis and anomaly detection.

While several methods exist for clustering, most of them involve several passes over the data. In data stream applications, systems should perform tasks in real time and usually do not have access to the entire data set. This means at each point in time they only have access to the most recent data and in some cases only to the most recent data item. To address this limitation, several clustering algorithms like partition based algorithms(like CluStream, StreamKM++), density-grid-based algorithms (like DStream-II,DenStream), Model-Based algorithms(like SWEM) were proposed that only make one pass over the data and in an evolving way.

In order to make streaming systems robust and scalable, current Data streaming systems incorporate a parallel architecture, where data streams are processed in several computing nodes simultaneously. This characteristic of the current systems add several complexities that any stream clustering algorithm should take into account. In addition to that, even though conceptually scalable, most of the current stream clustering algorithms have been tested on a single machine environment. Thus, to truly express the power of these algorithms, they must be

extended and tested in a distributed and parallel architecture after having dealt with the complexities of the system.

1.1 Motivation

With the revolution in hardware technology and increase in the use of sensors to record events and data, the need for processing such data which keeps on generating continuously has increased tremendously. Segregating this data in target groups have been an age old problem and clustering algorithms try to solve this problem by grouping samples together based on the commonality of the features. There are various models which follow different methodology for producing clusters which are discussed in detail later. Although there exists no single algorithm which is perfect for every situation and thus, producing good result depends on various other factors.

The data explosion in recent times has made the process of clustering more complex. Existing algorithms are not efficient anymore as there is practically no end to the data which is being generated. Even though algorithms like K-Means work well for small to moderate size datasets, it starts crumbling when we account for huge amounts of data. This is true due to the fact that iterative distance calculation is needed for every incoming data point against the cluster centers, and this procedure is the most time intensive task. To solve this issue, efficient methods of cluster creation were required which can work with data streams where data is practically infinite. Methods like these also need to scale really well so as to perform operations on the data in parallel, thus making the overall process even more efficient.

1.2 Goal

This thesis aims to validate the procedure of performing clustering on a data stream on a large scale. Our main goal was to perform survey of the existing stream clustering algorithm and then try and answer the following questions:

1. What are the different methodologies of performing clustering? What are the basic concepts which are involved?

2. Which of the algorithms are suitable for large scale clustering?
3. Which algorithms are suitable for clustering over a data stream?
4. As Flink is a comparatively new data processing system, how well does it behave when a machine learning algorithm is applied on top of it?

Apart from answering such questions, our goal is to pave a path towards implementing stream mining algorithms which is still not very popular but could be extensively useful.

1.3 Outline

This section deals with the workflow approach that was used to move towards the completion of the thesis and answer the questions which were mentioned in the previous section.The subsequent sections are organized as follows:

1. **Chapter 2** deals with the introduction to the concept of clustering data stream, its basic methodology and challenges associated with it.
2. **Chapter 3** gives an overview of all the algorithms which were reviewed belonging to various classes of clustering algorithms. Also, this chapter includes the comparative result of all the different algorithms reviewed previously.
3. **Chapter 4** deals with the idea behind selecting one of the algorithms for implementation reviewed in the previous chapter.
4. **Chapter 5** involves the implementation details where a brief introduction to flink systems is presented followed by explanation of the details of the algorithm implemented and the challenges associated with it.
5. **Chapter 6** includes all the experimentation done on the implemented algorithms and the results are compared against results from other data processing systems
6. **Chapter 7** outlines a brief overview of related work
7. **Chapter 8** concludes the thesis by explaining the results obtained and how could we proceed in the future

Chapter 2

Clustering Data Streams

Clustering is the process of grouping similar data points from a given dataset and is one of the widely used important data mining task. With the advancement in technologies and the data explosion, we have a lot of data but not enough information out of the data. Running data mining task is feasible on small to moderate sized data sources but when we consider large datasets then processing such huge datasets become practically infeasible. This is because such amount of data can't be stored in memory and processed subsequently. Also, the last decade has seen a consistent rise in usage of sensors, social networks, real-time data processing where data is continuously generated as data streams.

Performing data mining tasks on data streams is specifically challenging as most of the data mining algorithms assume fixed and finite datasets which can be physically stored and iterations can be performed over them. Similarly, data stream clustering is not as trivial as clustering on finite datasets as processing data streams inherently involve a lot of challenges, along with the challenges of the data mining algorithm. The challenges associated with data stream clustering would be explained in details in section 2.2.

2.1 Methodology

Data stream clustering algorithms basically involve the methodology of handling and processing the infinite amounts of data coming through the data stream which may evolve with time or finding a procedure by which data could be stored over

time and then processing it later. There are different methods of processing this infinite stream of data which are explained as follows [2] [3]:

1. **Store and process**
   In this process, incoming data streams are stored for a given amount of time and then clustering is performed over the stored data only. This is similar to a batch process although the accuracy gets affected as this process is done continuously. Also, it gets infeasible when the data stream is generating data quickly as the data mining task may take more time than the time it takes to store the incoming data and fill up the secondary storage. For the reasons discussed already, this method is not really popular with scientists and it is generally not used.

2. **One-Pass entire stream**
   In one-pass entire stream processing, the data is clustered by analysing the incoming data points only once. In this algorithm, the data points which arrive first becomes the representative of the first cluster and subsequent points become the representative of the other k-1 cluster in a k cluster scenario. Then, the points which come after the initial set of representative points are compared against the existing representative points and then it is assigned to the one which is determined to closest based on the pre-defined criteria. Finally, when the new point is assigned to a cluster, the representative point is updated.

   Based on the algorithm used, there could be new clusters which could be generated if the incoming point doesn't satisfy the pre-defined criteria for cluster assimilation or 2 clusters could be merged and the new point could be given its own cluster.

   The important thing to note in a one-pass entire stream processing method is that the clusters are the representation of all the points which has entered into the stream processing system so far.

3. **One-Pass evolving stream processing**
   In the one-pass evolving stream processing system, unlike the one-pass entire stream processing, the incoming infinite data stream is considered to be evolving with time. Hence, over a given period of time the results produced by the clustering algorithm may change, which means new clusters may appear over time and other may disappear. In scenarios like this, the evolving

data streams are processed in such a way that new data points are given greater importance than older data. For doing this, data stream clustering algorithms consists of defining windows which are used as updating blocks and which covers recent data so that the resulting clusters highlight the updates to the incoming stream rather than considering older data which might not be relevant anymore. For doing this, 3 different window models are available:

1. Landmark window 2. Sliding window 3. Damped window

4. **Online-Offline stream processing**
   Online-offline stream processing model is a two-step procedure where the first step involves keeping a summarized information of the incoming data stream using a one pass methodology explained previously. This helps in dealing with 2 challenges associated with the data stream:

   (a) It can process fast stream with comparative ease

   (b) It overcomes the memory constraints as we don't need to store any data points in the memory

   Thus, the online component helps in summarizing the dataset in real time so as to be processed later in the offline component. The offline component then performs clustering on the summary information. In practice, most of the stream clustering algorithm employ this 2-step architecture as it enables capturing dynamic data streams thus enabling detection of concept drift, at the same time the summary information is accurate enough to give good quality clusters.

2.2 Challenges

A good clustering algorithm needs to take care of the following restrictions related to data streams:

1. The data points are continuously pushed out and the speed of arrival of data depends on the data source, thus it could be really fast or similarly slow.
2. The data stream could be potentially infinite, which means that there could be no end to the incoming data.

3. The features and characteristics of the data points which are arriving may change over time.
4. The data points are potentially one-pass i.e. the data points can be used only once after which it is discarded, so fetching important characteristics of the data is really important.

Additionally, for creating good clusters, the clustering algorithms must have the ability to handle the following challenges before it could be used effectively:

1. **Ability to handle noise and outliers in the data** - A good clustering algorithm should address the problem associated with outliers as it tends to skew the formation of effective clusters. For this, the algorithms should reject outliers or involve a methodology to not get influenced by it.
2. **Ability to handle concept drifts** - This challenge is very specific to data streams and it is very important as well. This is true because a clear distinction between an outlier and a data point which is evolving with time needs to be made. The first one needs to be rejected whereas the latter needs to be accepted and a change in the concept needs to be acknowledged.
3. **Ability to handle fast data** - As data points from data stream arrive continuously, the clustering algorithm needs to manage and process them in real time as traditionally data streams could be huge and thus should be processed in one-pass.
4. **Ability to process data in given memory** - The huge amount of data shouldn't affect the processing capabilities of the stream. Thus, the clustering algorithm shouldn't need unlimited memory for the unlimited data points arriving in the system and it should be able to operate within the available memory.
5. **Ability to handle the curse of dimensionality** - With high dimensional data comes the additional problem of selecting the proper feature vectors or dimensions which could contribute to creating better clusters. Also, with high dimensional data comes the problem of additional calculation and additional processing thus the algorithm should take into account such factors so as not to be affected by it.

It should be mentioned at this time that none of the existing data stream clustering algorithms could solve all the aforementioned challenges. Therefore, the choice of the algorithm depends highly on the use case.

Chapter 3

Clustering Algorithm Survey

This chapter provides a brief yet comprehensive overview of the literature reviewed, this includes the various categories of clustering algorithms and a few important algorithms in each of these categories. These algorithms are divided into groups based on the methodology used in each one of them. As no single algorithms is perfect for all scenarios, the advantages of each category of algorithm depends on the problem which needs to be solved and the data which is available.

The 4 basic categories of clustering algorithms are Partition Based, Hierarchical Based, Density & Grid Based and Model Based. These categories are defined in the following section along with the algorithms which were studied and reviewed for the purpose of survey and comparison. Also, a summary closely related to each individual paper is presented, so as to present a concise yet fully understandable report. Later we also discuss the properties and comparison of all the algorithms reviewed and in Chapter 4 we select one of the reviewed algorithms for implementation.

3.1 Partition based

A partition-based clustering algorithm groups together incoming data points into a fixed number of partitions, where each of the partitions represents a cluster having the data points with similar characteristics and properties. These clusters are typically found using a distance measuring metric such as Euclidean, Manhattan etc.,

which calculates the physical proximity of each of the data point to the representative point of each cluster. These algorithms in effect are very similar to K-Means algorithm(put reference) but have been extended to work on streaming data. Some of the more popular algorithms in this category are STREAM which works on the LOCALSEARCH algorithm which itself is based on k-median search over data streams, along with this we have StreamKM++ which tries to solve the problem using concept called coresets and the very popular CluSTREAM algorithm which introduces the concept of online-offline processing framework described in 2. CluStream is also popular because a majority of the clustering algorithm adopts its concept of online and offline processing as it provides with a very innovative yet extremely useful method of clustering in addition to dealing with fast streaming data. The working model of these 3 algorithms are discussed in greater detail in the following sub sections:

3.1.1 STREAM

STREAM[4] is one of the older algorithms in this set and even though its performance is not at par with the more recent algorithms, it is essential to include this in the survey as STREAM motivated more research in the direction of stream clustering.

STREAM aims to cluster the incoming stream with 2 different algorithms

1. Using the local search[5] algorithm recursively
2. Extend the local search algorithm by relaxing the number of clusters in intermediate step called as LSEARCH

Clustering using Local Search

The clustering process maintains a forest of assignments where all the roots of the k trees are medians and the nodes inside the tree are points assigned to the median. Following is the list of steps of the algorithms described by the authors as is in [4]

1. Input the first m points; use a bicriterion algorithm to reduce these to *O(k)* (say *2k*) points. As usual, the weight of each intermediate median is the

number of points assigned to it in the bicriterion clustering. (Assume$m$ is a multiple of *2k*.) This requires *O(f(m))* space, which for a primal dual algorithm can be $O(m^2)$.

2. Repeat the above till $m^2/(2k)$ of the original data points are seen. At this point,m intermediate medians are obtained.
3. Cluster these m first-level medians into *2k* second-level medians and proceed.
4. In general, maintain at most m level-i medians, and, on seeing m, generate *2k level-i+1* medians, with the weight of a new median as the sum of the weights of the intermediate medians assigned to it.
5. After seeing all the original data points, all the intermediate medians are clustered into k final medians.

The number of levels require by the algorithm is at most *O(log(n/m)/log(m/k))* where m = $\sqrt{M}$ with M being the size of the memory.

For the bicriterion algorithm used in the first step of the algorithm, localsearch[5] is used so that the clustering is performed in space linear to m and thus, as the local search algorithm is quadratic in nature thus the total running time of the algorithm is dominated by the first step only.

To improve on the time complexity a subquadratic time approximation algorithm is used which is given by the authors[4] as follows

1. Draw a sample of size $s = \sqrt{nk}$.
2. Find k medians from these s points using the primal dual algorithm in [6]
3. Assign each of the n original points to its closest median.
4. Collect the n/s points with the largest assignment distance.
5. Find k medians from among these n/s points.
6. At this point *2k* medians are obtained.

Thus, algorithm which is provided above gives an *O(1)* approximation with *2k* medians having constant probability.

Improved Clustering using LSEARCH Algorithm

LSEARCH algorithms relax the requirement of k clusters in the intermediate steps thus this relaxation helps in reducing the quadratic nature of the local search[5]

algorithm described in subsection 3.1.1 and thus makes the overall algorithm run faster.

This algorithm is based on the facility location problem such that the number of k-medians is unrestricted but each additional center attracts an additional cost which should be taken care of.

Th goal of the facility location problem is to minimize the cost of facility clustering by selecting a value of k and a set of centers C, such that

$$FC(N, C) = z|C| + \sum_{i=1}^{k} \sum_{i=1} d(x, c_i)$$

where,
$d(x, c_i)$ is the distance of point x from center c_i,
z = parameter called facility cost

Also, given the facility location definition, the concept of gain needs to be explained which the cost one would save(or expend) if a facility needs to be opened at point x ∈ N.

The following algorithm is proposed by the authors to decrease the number of iterations from Θ(log n) to Θ(1) such that the best achievable cost is obtained faster.

Algorithm InitialSolution*(data set N, facility cost z)*

1. Reorder data points randomly.
2. Create a cluster center at the first point.
3. For every point after the first:
    - Let d = distance from the current point to the nearest existing cluster center.
    - With probability d/z create a new cluster center at the current point; otherwise, add the current point to the best current cluster.

Given the above described concepts, the complete algorithm consisting of Facility Location and LSEARCH is given as follows:

Algorithm FL*(N,d(.,.),z,∈,(I,a))*

1. Begin with *(I,a)* as the current solution.
2. Let C be the cost of the current solution on N. Consider the feasible centers in random order, and,for each feasible center y, if *gain(y)* > 0, perform all advantageous closures and reassignments (as per gain description), to obtain a new solution (I', a'). [a' should assign each point to its closest center in I'.]
3. Let C' be the cost of the new solution; if C' $\leq$ (1 - $\in$)C, return to Step 2.

And following is the LSEARCH algorithms which provides calculates the k-Median for a given distance function d and for a dataset of size N/

LSEARCH*(N,d(.,.),k,$\in$,$\in$',$\in$")*

1. $z_{min} \leftarrow 0$
2. $z_{max} \leftarrow \sum_{x \in N} d(x, x_0)$ (for x_0 an arbitrary point in N)
3. x $\leftarrow (z_{min} + z_{max})/2)$
4. (I,a) $\leftarrow$ InitialSolution(N,z)
5. Randomly pick $\Theta(\frac{1}{p}$log k) points as feasible medians
6. While #medians $\neq$ k and z_{min} ¡ (1 - $\in$")z_{max}
   - Let (F,g) be the current solution
   - Run FL(N, d, $\in$,(F,g))to obtain a new solution (F',g')
   - If k $\leq |F'| \leq 2k$, then exit loop.
   - If $|F'| > 2k$, then $z_{min} \leftarrow$ z and z $\leftarrow (z_{min} + z_{max})/2)$; else if $|F'| < k$, then $z_{max} \leftarrow$ z and z $\leftarrow (z_{min} + z_{max})/2)$
7. Return solution (F', g'), where each cluster is moved to the center of the mass of its cluster to simulate a continuous space

The entire LSEARCH algorithm can be summarized as the process of continuously creating cluster centre with a calculated probability and then reduces the number of cluster till we have the target number of medians.

Also the running time of LSEARCH is O(nm+nklogk), where m is the number of facilities opened InitialSolution.

Merits and Limitations

Merits

- One of the first algorithms to solve clustering over streaming data.

Limitations

- Difficult to comprehend and execute.
- No concept drift detection.
- Outlier detection not very clear.
- Problems in dealing with large amount of high dimensional dataset.

3.1.2 CluStream

Even though CluStream[7] is as old as STREAM[4], still CluStream is more popular as it employs a very unique online-offline framework. The entire clustering process in CluStream is divided into an online micro-clustering component which requires a very efficient process for storage of statistical summary of the incoming data stream and an offline macro clustering component which uses the summarized data in order to provide cluster information as and when required by users. This framework is widely used in many stream clustering algorithms because of its efficiency in handling data streams.

Clustream Framework

Micro-clustering is an innovative concept to summarize data streams efficiently and record temporal locality of the data in the data stream at the same time.

For the online process of storing statistical summary, the concept of microclusters are used, which is similar to the cluster feature vector used in BIRCH [8] with an extension to store the temporal information as well. This summary information which are stored in the micro-clusters are used by the offline component, which in itself is dependent on the user inputs.

These microclusters needs to be stored at specific snapshots such that there is an effective trade-off between the storage requirements and the ability to fetch the statistical summary from different time horizons. This is achieved by storing the snapshots using a pyramidal time frame, so that the cluster statistics could be analysed from different periods of time.

The concept of microclusters is defined by the authors of CluStream as follows:

A microcluster for a set of d-dimensional points $X_{i_1}...X_{i_n}$ with timestamps $T_{i_1}....T_{i_n}$ is defined as the (2.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.

where the entries of the tuple is defined as follows:

- $\overline{CF2^x}$ - It maintains the sum of squares of data values for each dimension. Thus, it contains d values where the pth entry of $\overline{CF2^x}$ is equal to $\sum_{j=1}^{n} (x_{i_j}^p)^2$
- $\overline{CF1^x}$- It maintains the sum of the data values for each dimension. Thus, it also contains d values where the pth entry of $\overline{CF1^x}$ is equal to $\sum_{j=1}^{n} x_{i_j}^p$
- $CF2^t$ - the sum of squares of timestamps $T_{i_1}...T_{i_n}$
- $CF1^t$ - the sum of timestamps $T_{i_1}...T_{i_n}$
- n - the number of data points

The summary information can be expressed in an additive way over the different data points thus making it a natural choice for use in data stream based clustering algorithms. Also, the micro-cluster for a set of points $\mathcal{C}$ is given by $\overline{CFT(\mathcal{C})}$

Online Phase

The online phase of the algorithm is not dependent on any user inputs and is aimed to maintain statistical summary of the incoming data points at a sufficiently higher level of granularity for both temporal and spatial information. The online phase also is also the phase where micro-clusters are formed.

The online phase maintains a total of q micro-clusters at any moment of the algorithm execution. This value q is determined by the available main memory for micro clusters storage. Thus, it is generally significantly larger than the natural number of clusters in the data but also significantly smaller than the total number of data points, which may be conceptually infinite in streaming sources.

At the start of the algorithm execution, P points are stored on disk from the data stream and the standard k-means algorithm is applied to create q micro-clusters.

After the aforementioned step, the updation process of the created microclusters are started. So, whenever a new data point arrives to the system, any 1 of the following 2 possibilties takes place:

1. The point is absorbed by one of the existing micro-clusters based on the closeness of the cluster to the data point, this generally is calculated using the distance metric using the centroid of the micro-cluster and the data point and the data point is absorbed to the nearest micro cluster using the additive property.

2. The point is placed in its own micro-cluster, but the number of micro cluster is fixed. Thus, the number of other clusters needs to be reduced by one which can be achieved by either deleting one of the older micro-cluster or merging 2 microclusters together.

For case 2 where a point is given its own microcluster, relevance timestamp of cluster M is calculated using the timestamp information stored in the microcluster.So, if for a given micro cluster M, the least relevance stamp is lesser than a user-defined threshold, that micro-cluster is eliminated and a new micro-cluster is created with an unique cluster id. In case when the relevance timestamp is greater for all the available microcluster, it means that all the micro-clusters are comparatively new and the value of the relevance timestamp is greater the user-defined threshold. In such cases, 2 closest microclusters are merged and the new data point is given its own micro cluster.

Case 2 here is specifically helpful in understanding and capturing concept drift in the stream.

Offline phase

The offline phase takes into account the compactly stored summary available in the micro-clusters, thus the constraint of the stream that only one pass over the data is available does not hold in this scenario.

For the input of the offline phase, the user supplies the number of high level clusters k which needs to be calculated and the time horizon h over which the clustering needs to be performed. This choice determines whether the cluster formed are more detailed or they are rough. The use of pyramidal time frame comes in handy in this case as it ensures the availability of snapshots which can be used to calculate approximate micro-clusters for a defined time horizon. The micro-clusters obtained after applying pyramidal time frame are then clustered

on a high level to createa smaller number of clusters, with each cluster having a group of microclusters assigned together, thus each microcluster is treated as pseudo points for the macro clustering algorithm.

Merits and Limitations

Merits

- Very straightforward to understand and implement.
- Novel approach of capturing temporal activity to cluster feature helps in achieving clustering over any time horizon.
- The use of online and offline phases helps in capturing essential statistical information quickly and then generate good clusters.

Limitations

- Clustream is partition based and as any general partition based algorithm, it is sensitive to outliers.
- The micro-clusters and the macro-clusters generated are spherical in shape which might not be the case always.
- Dealing with high dimensional data would be slower as it is still dependent on euclidean distance calculations.

3.1.3 StreamKM++

StreamKM++[9] is the last reviewed algorithm from the partition based concept and it takes a sample of the data points from the data stream and computes a small weighted sample of it.It then solves the problem of clustering on the sample using the K-means++[10] algorithm. To compute this small sample, the authors propose two novel techniques.

1. Using an approach similar to the K-means++ seeding procedure, a non-uniform sampling of the incoming data points is done to obtain a small coresets. This procedure has a running time which does not depend a lot on the dimensionality of the data.

2. A new data structure called "a coreset tree" is proposed by the authors which significantly speeds up the process of non-uniform sampling required during the coreset construction process.

To understand the process of using coreset tree, the following few concepts need to be very briefly:

Coreset Construction

Let P be a set of initial n points from which a set of m points need to be chosen denoted by S. Thus, the coreset construction can be defined as the procedure of creating a weighted set over S' by selecting the set of m points (given by S) using the KMeans++ seeding procedure.

S' is obtained by using a weight function on the points from P which are close to points within set S.Thus, the set S' denotes the coreset for the StreamKM++ algorithm. This entire procedure of constructing coresets depends linearly on dimension d and as explained by the authors, pretty easy to construct as well.

Coreset Tree

The problem of using Kmeans++ seeding procedure to select the m sample points in S is that for each subsequent point, the distance of each point in P needs to be calculated to its nearest neighbour in S, thus increasing the computation time when the value of m is very high as it takes $O(d.n.m)$ time to obtain m coreset points.

This was the reason why the concept of coreset tree was created, as it enables to compute the points in the sample S by taking points from a subset of P only which is significantly smaller than n. Thus needing a time of the order of $O(d.n.logm)$ to compute all m coreset points, provided that the coreset tree is balanced.

Before explaining the procedure of construction, the coreset tree needs to be defined formally. Thus, a coreset tree T is in essence a binary tree which can perform hierarchical divisive clustering on a given set of points P.So, the process of clustering starts from a single cluster which is then followed by successive partition into

two cluster and this continuous partitioning process is repeated until the desired number of clusters is reached.

As explained by the authors of streamKM++,the coreset tree, T needs to satisfy the following properties:

- Each node of T is associated with a cluster in the hierarchical divisive clustering.
- The root of T is associated with the single cluster that contains the whole point set P .
- The nodes associated with the two sub-clusters of a cluster C are the child nodes of the node associated with C.

Also for every node v of T, the following attributes needs to be stored:

- A point set P_v which is the cluster associated with node v, which is explicitly stored only in the leaf nodes,
- A representative point q_v from set of points P_v obtained by sampling using Kmeans++ procedure
- An integer $size(v)$ which represents the number of points in set P_v and
- A value $cost(v)$ for the leaf node which is the sum of squared distances over all points in P_v to q_v, whereas for the inner nodes its simply the cost of its children.

The description of the structure of the coreset tree leads to the next step, which is of constructing the tree.

Construction - Coreset Tree The construction of tree T starts with a single node, the root, which is associated with all the points in the set P. From this set, a set of m sample points needs to be computed. Assuming that the current tree has i leaf nodes, the next sample q_{i+1} can be obtained which would be a new cluster and a new node in the tree T using the following methods:

1. Choose a leaf node l at random.
2. Choose a new sample point denoted by q_{i+1} from the subset P_l at random.
3. Based on q_l and q_{i+1}, split P_l into two subclusters and create two child nodes of l in T .

The update in the leaf nodes is then propagated upwards until the root of the tree is reached.

Coreset Creation Methodology

As the coreset tree has been constructed, the leaf nodes of the tree represented by the sample points $q_1, q_2, ...q_m$ gives us the the coreset S, where the weight of each sample point q_i is given by the number of points for each of the leaf nodes in the tree. Given that, the algorithm maintains a size of L buckets for the datastream containing n points where $L = \left\lceil log_2(\frac{n}{m}) + 2 \right\rceil$ Following are properties of each bucket:

1. Bucket B_0 can store any number of points between 0 and m, whereas Bucket $B_i (i > 0)$ is either empty or have exactly m points.
2. At any point, if a bucket B_i is full, it consists of a coreset of size m representing $2^{i-1}m$ points from the data stream.

Accordingly, the bucket filling procedure depends on the following conditions:

- New points from the data stream are always inserted in bucket B_0.
- If bucket B_i is full, all points are moved from B_i to B_{i+1}.
  - If B_{i+1} is empty, nothing is done
  - If B_{i+1} is full, compute new coreset of size m using the coreset construction methodology, from the union of B_i and B_{i+1}.

StreamingKM++ Algorithm

As the procedure of obtaining the coreset has been clearly described, thus the entire algorithm can be presented for data stream. The algorithm is described as follows:

1. Extract a small coreset of size m from the data stream by using merge and reduce.
2. For the reduce step, get the coresets using the coreset trees by employing the coreset construction methodology described in subsection 3.1.3.

3. Run any kmeans algorithm on the coreset of size m, as its smaller than and even independent of the size of the data stream, thus the results are accurate and does not depend on the original data points.

Merits and Limitations

Merits

- Comparatively easier to understand and implement.
- Claims to be really fast in regards of creating the coreset tree.

Limitations

- Needs to know the size of the stream (Number of data points) to compute the number of buckets to be used.
- Not feasible for unlimited size streams
- No drift detection methodology, clusters created over the entire stream.

3.2 Hierarchical based

Hierarchical Clustering: A hierarchical clustering method groups the given data into a tree of clusters which is useful for data summarization and visualization. The following algorithms attempts to do hierarchical clustering on streaming data.

3.2.1 BIRCH

BIRCH[8] is the oldest algorithm reviewed in this survey and it claims to deal with large datasets by generating a more compact summary and then use this summary for clustering. Thus, a singe scan of the data is enough to get decent clustering result. This property is specifically useful in case of streaming data as essentially the incoming data from streams are not generally stored as streams could be potentially infinite and multiple passes over the data is generally time consuming or simply not possible.

Clustering Feature and CF-tree

BIRCH summarizes the incoming data points into a set of sub clusters called Clustering Feature(CF) to reduce the scale of the clustering problem.

Clustering Feature(CF) A CF entry is a triple which contains the summary information of the incoming data points. It is given as follows:

CF = (N,$\overrightarrow{LS}$,SS)
where,
N = number of data points in the cluster
$\overrightarrow{LS}$ = Linear sum of N data points SS = Squared sum of N data points

The Cluster features have additive property such that 2 cluster features CF1 and CF2 can be merged to create a subcluster with cluster feature CF as follows:

$$CF = CF1 + CF2 = (N1 + N2, \overrightarrow{LS1} + \overrightarrow{LS2}, SS1 + SS2)$$

Similarly, it has subtractive property which follows the same intuition.

A CF entry for a sub-cluster is not only compact but it also provides accurate results as it is sufficient for calculating all the measurements which are needed for BIRCH.

CF Tree A CF tree is a height-balanced tree with two parameters:

1. Branching factor,B for inner node and L for leaf node
2. Threshold T

Each inner node contains at most B entries of the form [CF_i, $child_i$], where i = 1,2, ...,B. The $child_i$ in the node is a pointer to the $i-th$ child node in the tree and CF_i is the CF entry of the sub-cluster given by the $i-th$ child.

Both the inner node and the leaf node represents a sub-cluster made up of all the sub-cluster represented by its entries. A leaf node can have at most L entries where all the entries are CF's in themselves. Also, every leafnode are constituted of 2 pointers 'prev' and 'next', so that sequential scans along the leaf ndoes are

efficiently done. But at the same time, all entries in a leaf node must adhere to a threshold T, i.e. the diameter/radius of each leaf entry has to be less than T.

As, the structure of the tree has been defined, the procedure of inserting a data point or an existing subcluster in the tree needs to be defined. This is briefly discussed as follows:

1. **Leaf identification**: Find closest closest child node
2. **Leaf modification**: Absorb the data point or subcluster to the nearest leaf element such that the threshold criteria T is not violated. If the threshold criteria is not met, then a new entry needs to be created in the lead node provided there is enough space for a new entry, else the leaf node is split using the farthest pair of entries and the remaining entries are arranged in the 2 leaf nodes based on the distance criteria.
3. **Leaf path modification**: When a new entry is inserted to a leaf node, the information in the CF is updated on the entire path from the root to the leaf. This is specifically simple if the new entry is simply absorbed, but in the case when the leaf is split into two, then the information about the new leaf node needs to be inserted into the parent node. This new entry in the parent non-leaf node could in turn cause a split if not enough space is available in the inner node. This behaviour in general could be propagated all the way to the root, and if the root needs to be split as well then in that case the tree height increases by 1.
4. **Merging Refinement**: As page sizes are responsible for causing splits thus,if the order of the incoming data is skewed, then in that case the CF tree could get skewed and it could have a bad effect on the overall quality of the clusters. To solve this issue an additional merge can be performed in the inner nodes where the leaf split propagation ends. This inner node with the new entry corresponding to the leaf split can be scanned for 2 entries which are the closest to each other and merge them. The corresponding merge also results in merging the corresponding child nodes thereby increasing space for one more entry in the inner node and also increasing page utilisation.

Overall BIRCH algorithm

The BIRCH clustering algorithm consists of 4 phases, a couple of which are optional and are used solely to improve the results. Although, its important to note

that these optional steps may require additional passes over the data which is not applicable for true streaming data.

1. **Loading** - This phases scans all the incoming data and builds the initial CF-tree in-memory, subject to the available memory. Thus, this phase essentially creates the summary of the incoming data point. This is the most important phase of BIRCH, as this reduces the problem of clustering data points to clustering subclusters present in the leaf entries. This is important because clustering the subclusters are essentially much faster than performing clustering on the datapoints. Also, this phase removes the outliers as well, thus the resulting clusters are more accurate.
2. **Optional Condensing** - This phase is optional and performs the condensing of the intitial CF-tree by rebuilding a smaller CF-tree. It achieves this by removing more outliers and condensing the subclusters which are close together into larger ones.
3. **Global Clustering** - This phase performs the clustering of the resulting CF-tree either from phase 1 or after the condensing from phase 2. Thereby obtaining the different patterns and structures available in the data.
4. **Optional refining** - This phase is also optional but it aims at refining the clusters which were created in phase 3 by using the cluster centres as seeds and then redistributing the data points to the closest cluster centres, thus obtaining a refined result and a better set of clusters. It needs to be noted that another pass over the data is done which is generally not possible with pure and infinite streams.

Challenges

The limitation of each node to hold only a fixed number of entries introduces 2 major challenges in BIRCH. They are:

1. Two subclusters which should be in the same cluster are split into different nodes and similarly two subclusters which shouldn't be together are part of the same node. This challenge is solved by using an already existing algorithm (global or semi-global) for clustering all the entries in the leaf node across the different leaf nodes, thus such anomalous entries would not effect the final clustering.

2. In case same data point is inserted twice, the resulting cluster for both of them could be different as they might be put into different nodes. This challenge is solved by using the phase 4 of the BIRCH algorithm where clusters are refined by readjusting the data points according to the clusters. This challenge is thus not easily solvable because the data points are not available for a second pass in pure streaming scenarios.

Merits and Limitations

Merits

- The first algorithm which proposed to store summary information as Cluster Features(CF), used extensively in other algorithms.
- Can handle large amount of data.
- Essentially one pass algorithm, after creating CF-tree, any global or semi-global algorithm could be applied.

Limitations

- Cluster quality dependent on a second pass which in most cases is not available.
- Does not detect concept drift as no concept of decay of older data.

3.2.2 ClusTree

ClusTree[11] claims to be a compact and self adaptive index structure for maintaining the summary of the data coming via stream. In this algorithm the authors propose a parameter free algorithm which is capable of processing the stream in a single pass and with available memory. It also dynamically adapts to the speed of the data stream by using a concept called anytime inserts. Correspondingly, anytime clustering approach is proposed which makes the algorithm capable of delivering a result at any given point in time. It uses a strategy similar to CluStream [7] to forget older data and give more importance to new data points. Additionally, the authors propose novel descent strategies to improve the clustering result on slower streams as long as time permits.

ClusTree procedure with anytime inserts

ClusTree creates and maintains compact summaries of incoming datastream using a popular technique known as Micro-Clusters which is already explained in BIRCH and CluStream algorithms. Thus instead of storing all incoming objects, it maintains a cluster feature tuple similar to BIRCH, given by CF = (n, LS, SS) where n = number of points, LS = Linear Sum and SS = Squared sum.

The problem with traditional microclustering approach is that it lacks support for varying stream speeds. The authors of ClusTree propose extending the index structures available in the R-tree family to maintain CF's. This allows maintaining a hierarchy of microclusters at different granularity levels. Thus, if a given data point reaches the leaf node and that the data point is similar to the microcluster then it is absorbed by it, else a new micro-cluster is created.

The problem with such hierarchy is that there might not be enough time to insert the object at the leaf node as the stream speed varies and time taken to process one data point might be more than the speed at which the next data point comes. Thus the concept of anytime inserts is used. This allows to store the incoming data point temporarily in a local aggregate, so that necessary information for clustering is maintained and rather than discarding, the newly arrived data point is inserted somewhere. The advantage local aggregates provide over local queues is that the time for regular inserts can be used to take a buffered local aggregate to the corresponding leaf node as a hitchhiker by a descending data point.

Clustree

The definition of ClusTree is provided as is from [11] and is given as follows:
A ClusTree with fanout parameters m, M and leaf node capacity parameters l, L is a balanced multi-dimensional indexing structure with the following properties:

- an inner node nodes contains between m and M entries. Leaf nodes contain between l and L entries. The root has at least one entry.
- an entry in an inner node of a ClusTree stores:
  - a cluster feature of the objects it summarizes.
  - a cluster feature of the objects in the buffer. (May be empty.)
  - a pointer to its child node.

- an entry in a leaf of a ClusTree stores a cluster feature of the object(s) it represents.
- a path from the root to any leaf node has always the same length (balanced).

The tree defined above is created and maintained like any multidimensional index such as R-tree, R*-tree etc. For insertion, the subtree with the closest mean with respect to euclideam distance is chosen.

Here, its important to mention that the buffer in each entry of the tree is really crucial as that shows the anytime capability of ClusTree. It is used as a temporary storage for the data points when the insertion procedure is interrupted and the data point couldn't reach the leaf node on time. Thus, whenever a future access to this subtree is made, the the temporary buffer entry is taken along as hitchhiker to the leaf node. Interestingly, whenever moving down the subtree, the destination differs for the original data point and the hitchhiker, the hitchhiker is placed in the buffer of the corresponding split node, such that some other data point may carry it down.

This concept of hitchhiking and temporary buffer storage is crucial for ClusTree's anytime clustering capability.

Cluster updates and decay

Similar to CluStream, ClusTree uses a decay rate λ which controls how much weight the new items have over the old items. Thus higher the value of λ, faster is the process of forgetting the older data. For this, the data points are weighed with an exponential time-dependent decay function,

$$\omega(\Delta t) = \beta^{-\lambda \Delta t}$$

where $\beta = 2$ (value chosen by the authors as optimum).

As for any algorithm which incorporates decay factor, ClusTree also has to add temporal information to the nodes of the tree. It is ensured that the inner nodes of the tree summarizes their corresponding subtrees accurately by making elements of a cluster feature vector dependent on the current time t, thus:

$$n^t = \sum_{i=1}^{n} \omega(t - ts_i)$$

$$LS^{(t)} = \sum_{i=1}^{n} \omega(t - ts_i).x_i$$

$$SS^{(t)} = \sum_{i=1}^{n} \omega(t - ts_i).x_i^2$$

where n = number of contributing objects ts_i = timestamp at which data point x_i was added to CF

As the additive property of CF and temporal multiplicity are preserved (add reference 3). Thus, if no object is added to a CF during the time interval [t, t+ Δt], then

$$CF^{(t+\Delta t)} = \omega(\Delta t).CF_{(t)}$$

Now that the decay procedure has been explained, it is useful to note that data point x carries the timestamp t_x of its arrival. Knowing this the procedure of updating entries in the nodes are explained as follows:

1. Each entry in a node has a timestamp $e_s.t_s$ which specifies the time of last update.

2. When a data point descend to a node in the tree, all entries of e_s in the node is updated to timestamp of the arrived data point, t_x by position wise multiplication with the decay function.
   $e_s.CF \leftarrow \omega(t_x - e_s.t_s).e_s.CF$
   $e_s.buffer \leftarrow w(t_x - e_s.t_s).e_s.buffer$

3. The timestamp is reset to the timestamp carried by the incoming data point, thus $e_s.t_s \leftarrow t_x$

It is important to specify here that all the entries in the same node gets the same overall timestamp as all entries are updated in the node in which the data point descends to.

The above procedure of weighing with time, helps in avoiding splits and saving time. Thus, if a node is about to be split, the algorithm checks whether the

least significant entry can be discarded or not. Assuming that a snapshot of the ClusTree is taken regularly after t_{snap} time, the significance is tested by checking whether the entry $\hat{e}$ with the smallest $n_{\hat{e}}^{(t)}$ satisfies

$$n_{\hat{e}}^{(t)} < \beta^{-\lambda.t_{snap}}$$

If this is the case, $\hat{e}$ is discarded, making room for the incoming data point to be inserted, and avoiding a split. The summary statistics of $\hat{e}$ are subtracted from the corresponding path up to the root. Also, its important to note that no entry is discarded if a new object has been added to it after the last snapshot has been taken.

Aggregation

The problem with fast streams is that insertion of data points would be interrupted continuously with the data points being stored in the root or upper level of the tree with little chance of getting down the tree. The worst case is of merging of dissimilar data points in the buffer which becomes inseparable in the buffer and thus having very bad cluster quality.

To solve this issue, the authors propose a speed-up through aggregation before insertion. This is achieved by not inserting each data point individually. Rather this is achieved by adding up m incoming data points and inserting the aggregate and summing up the next m data points.

The problem here is that very dissimilar data points can go to the same aggregate and we get the similar scenario of buffer back. To solve this, the authors keep several aggregates for dissimilar objects and make sure that the objects summarized in the same aggregate is similar.

To perform this segregation, a max radius for the maximum distance of the data point in the aggregate is set. The value or max-radius is determined from the leaf level as the average variance of the leaves. Thus, fast streams don't deteriorate the quality of the cluster very bad.

Also, the number of aggregates to be maintained is determined by the speed of the stream, i.e. it cannot exceed the number of distance computations that can

be done between two arriving items. In the case of a varying data stream, the maximum number of aggregates has to be set by the user.

Time Utilisation with Slower Streams

The default mode of working with normal speed streams is to reach the leaf node by always picking the child with the respective smallest distance between the data points and the children reachable from the current node. This can be considered as a single-try depth first approach.

When the stream speed is slower than usual, using a depth first approach would leave the algorithm idle once the data point reaches the leaf node. Thus, the authors explore alternative ways of choosing paths down the tree, as well as ways on how to spend any time that might still be available after reaching leaf level due to small model size and variation in object inter-arrival time.

Priority breadth first traversal The depth first strategy does not perform any kind of backtracking and hence cant correct any misguided choice that might occur due to overlapping entries on higher levels of the tree.

Thus to find the closest entry breadth first search evaluates all entries per level by sorting the entries by the distance distance of their corresponding parent to quickly find the closest option.

The advantage of this procedure is that as the entries are sorted according to the distance, the likelihood of finding the closest maintained micro-cluster within the first few entries of the priority queue on the leaf level increases. The only difference from depth first strategy is that the entries on the final path are updated at the time of interruption and not as the data point goes down the path.

Best first traversal As the depth first approach uses a greedy approach to descend down the tree, and it is not able to revise its choice, thus misleading aggregate information on the upper level could lead to bad clustering quality as it is possible for the micro-clusters in the lower levels to have a comparatively high distance to the data point when compared to nearby leaves.

Thus to handle such situations, best first strategy maintains a priority queue of the entries seen so far along with the corresponding distance to the data point to be inserted. Given the time to make the next step in descending down the tree, the best first strategy always takes the first element from the queue i.e. the entry which has the smallest distance to the object.

This is followed by calculating the corresponding distances of the data point to the child node and insertion of the entries in the priority queue. This process is repeated until interruption or all the nodes have been visited.

The best first strategy means that the decision which node to be inserted in the priority queue is now based on all the information that the algorithm has at the time of the decision making. The next descent is made along the path given by the priority queue even if it means not continuing the deepest path. Thus best first can be considered as a global optima strategy compared to the depth first which reaches for a local solution.

Iterative depth first descent A possible drawback of Best first and priority breadth first is that in both the algorithms its possible for the data points to be buffered at a higher level of the tree depending on how soon the process is interrupted, in comparison the depth first descent makes the best case to reach the leaf nodes but it compromises on the quality.

The iterative depth first descent tries to take the best of all worlds by initially following the depth first descent strategy but then upon reaching the leaf level, it iteratively evaluates the decisions taken at the nodes on the depth first path as long as time permits.

The algorithm after reaching the leaf level goes back to the root and then descends into the sibling of the entry chosen during the last iteration. This gives two more alternatives along with the already available leaf node and the best is chosen out of the three. If there is still no interruption, the process is again repeated and continued until the algorithm is interrupted or no more unchecked siblings remain in the path. On interruption, the usual method of buffer and update is implemented as with the other algorithms.

Thus all these alternatives use the available time in the best possible ways and help in improving the quality of the cluster.

MacroCluster generation

The process of inserting the data points in the tree and storing results in the corresponding leaf levels can be seen as the online component of the algorithm. Thus, the microclusters stored in the leaf levels with the cluster features can be used in the offline macroclustering phase where the means of the CF's could be taken as representative points and k-means clustering could be applied or a density based clustering algorithm could be applied as well to detect clustering of arbitrary shape.

One of the main advantages of ClusTree is that it can maintain a large nummber of microcusters compared to other algorithms like ClusStream and hence the offline macroclustering procedure has more finer granularity of input which is specifically useful for density based clustering approaches.

Merits and Limitations

Merits

- Anytime clustering and self adaptive model size using buffer and hitchhiker concepts.
- Improved clustering quality on very fast streams using aggregation.
- better usage of idle time by using alternative descent strategies.
- Microclusters can be used to create macro cluster using other popular algorithms.

Limitations

- Limited applicability to density-grid based clustering algorithms as microclusters might not resemble dense grids exactly, thus might have difficulty in generating clusters of arbitrary shape.

3.2.3 ODAC

ODAC[12] is functionally different from the other algorithms reviewed in the survey. It clusters not the incoming data points but the attributes. Thus, in ODAC

each attribute is a time series and a data point which is fed to the system is the combination of observation from all time series at a particular time. The ODAC system uses Pearson's correlation coefficient [13] as a dissimilarity measure between time series over a data stream. This is followed by an agglomerative phase which helps in detection of concept drift by capturing the dynamic changes in the stream.

Unlike most partition based algorithms, in hierarchical clustering one doesn't need to know the value of k (Number of clusters) in advance. Thus hierarchical structures presents a better way to solve the problem of clustering . The are two known strategies for hierarchical clustering, divisive and agglomerative. This paper incrementally uses the divisive approach to create a hierarchy of clusters using the aforementioned dissimilarity measure.

ODAC overview

The process of clustering is simple and it consists of a top-down approach where one starts with all the data points as a single cluster and then incrementally split the clusters based on the diameters of the clusters. The leaf level nodes of such tree are the final clusters produced by the algorithm. It should also be ensured that the overall intra-cluster dissimilarity is being reduced after every split of the clusters.

For the calculation of similarity score between time series, Pearson's correlation coefficient is used which is defined as follows:

$$corr(a,b) = \frac{P - \frac{AB}{N}}{\sqrt{A_2 - \frac{A^2}{n}}\sqrt{B_2 - \frac{B^2}{n}}}$$

where,
$A = \sum a_i, \quad B = \sum b_i, \quad A_2 = \sum a_i^2, \quad B_2 = \sum b_i^2, \quad P = \sum a_i b_i,$
n = total number of data points

Thus, given the correlation coefficient, the dissimilarity between a and b is given as follows,

$$rnomc(a,b) = \sqrt{\frac{1 - corr(a,b)}{2}}$$

where rnomc(a,b) = Rooted Normalized One-Minus-Correlation

ODAC implementation

The splitting of a leaf node in the hierarchy depends on the minimum number of observations necessary to ensure convergence so that the leaf node can be tested for splitting. This is done in ODAC using Hoeffding bounds [14] which is preferred over using a user-defined parameter as it is independent from any probability distributions which generates the observations. Thus the Hoeffding bound helps in selecting the pair of variables which represents the diameter of the cluster.

Having the stated principles in place, the ODAC system feeds in new data points and they are processed only once. Also, the dissimilarity matrix is only computed for each leaf only when the true diameter is known with confidence given by the Hoeffding bound, this helps in speeding up computations as for every new data point only the leaves are updated.

Once the leaves have saturated enough with incoming data points such that it can be tested for splitting or aggregation, then the following heuristic is used for splitting the cluster Ck,

$$(d_1 - d_0)|d_1 + d_0 - 2\overline{d}| > \epsilon_k$$

where,
d_0 = minimum distance between variables in the cluster respectively
d_1 = distance between the pair of variables with maximum dissimilarity
$\overline{d}$ = average of all distances in the cluster

It may happen that the decision of splitting which created a certain leaf may be outdated owing to the change in the structure of the underlying data stream (concept drift). To capture this change, the aggregation of the leaf nodes may be done. For calculating this, the Hoeffding bound could be applied, such that a cluster C_k can be aggregated on the parent C_j, along with the sibling C_s, if

$$2.diam(C_j) - (diam(C_k) + diam(C_s)) < \epsilon_j$$

Thus if the above condition satisfies, the child nodes merge to the parents and the total number of cluster reduces and the new leaf starts new computation due to the underlying concept drift of the data stream.

Merits and Limitations

Merits

- Attribute clustering gets benefited from distributed systems, as different attribute streams can be handled separately.
- Number of clusters need not be fixed beforehand
- Constant time and space complexity

Limitations

- Noise detection not defined in the paper
- Paper not very detailed

3.3 Density Grid Based

Density-Grid Based Clustering: Density based clusters are defined as areas of higher density than the remainder of the data set. Objects in the sparse areas - that are required to separate clusters - are usually considered to be noise and border points whereas Grid-based clustering is independent of distribution of data objects. In fact, it partitions the data space into a number of cells which form the grids. Grid-based clustering has fast processing time since it is not dependent on the number of data objects. The following algorithms attempts to do density-grid clustering on streaming data:

3.3.1 D-Stream I

The D-Stream I[15] algorithm uses an online component similar to that used in clustream, which maps each input data into a grid and a corresponding offline component which computes the density of the grids and subsequently cluster the

grids based on the density. The algorithm also employs a density decay factor to capture the changes in the structure of the data dynamically.

The algorithm aims to solve 2 main challenges associated with clustering data streams:

1. The algorithm aims to capture dynamically changing data, thus it should be able to detect changes in the structure of the data over time and create corresponding clusters.
2. As we are dealing with large amounts of streaming data, it is impossible to retain the density information for every data point.

D-Stream Procedure

The D-Stream algorithm processes data points from data stream according to a discrete time step model. Thus,

1. At each time step, The online component of D-Stream algorithm keeps on reading the data from the stream and place the data into a density grid in the multidimensional space which corresponds to the dimension of the data, followed by updating the characteristic vector of the density grid.

2. At each gap time step, the cluster is adjusted by the offline component dynamically. So, after first gap time step, the initial cluster is generated after which the algorithm periodically removes the sporadic grids and new clusters are created.

Density Grids

The density grids is the result of partitioning the space S represented by the d-dimensional data. The space is represented as follows:

$$S = S_1 \times S_2 \times .....S_d$$

Now for each dimension, its space S_i where i = 1,....d is divided into p_i partitions as follows:

$$S_i = S_{i,1} \cup S_{i,2} \cup ..... \cup S_{i,p_i}$$

Thus, the data space S is partitioned in N density grids where $N = \prod_{i=1}^{d} p_i$

Subsequently, a data record $x = (x_1, x_2....x_d)$ can be mapped to one of the density grids as,

$$g(x) = (j_1, j_2, ...j_d) where x_i \in S_{i,j_i}$$

For each of the data record x, a density coefficient is assigned which decreases with the increase in time.The density coefficient at time t is given as follows,

$$D(x,t) = \lambda^{t-T(x)} = \lambda^{t-t_c}$$

where T(x) is the timestamp when data point x arrives at time t_c and $\lambda \in (0,1)$ is a constant called the decay factor.

Grid Density The grid density D(g,t) is defined as the sum of all the density coefficient of the data point that are mapped to the density grid, g. Thus, the density of g at t is given as follows:

$$D(g,t) = \sum_{x \in E(g,t)} (D(x,t))$$

The density of the grid can be updated whenever a new data point comes to the grid, for this purpose the timestamp for the last data point needs to be recorded. Thus, the updation procedure of density grid g can be shown as follows:

$$D(g,t_n) = \lambda^{t_n - t_l} D(g,t_l) + 1$$

where,
t_n = timestamp of the new record and t_1 = timestamp of the last record

This above procedure allows to update a single grid out of the N grids leading to a O(1) running time. Also, only the last 2 timestamps needs to be stored, thus saving space as well.

Characteristic Vector It is defined a tuple $(t_g, t_m, D, label, status)$ for a given grid g where, t_g = last time when g was updated

t_m = last time when g was removed from grid_list as a sporadic grid(if ever)
D = Grid Density at time t_g
label = class label for the grid
status = SPORADIC, NORMAL

Grid types As the sum of the density coefficient of all the data records which are part of the system will never exceed $\frac{1}{1-\lambda}$, thus the average density of every grid can be not more than $\frac{1}{N(1-\lambda)}$

These observations gives rise to the following definitions:

Dense Grid:

$$D(g,t) \geq \frac{C_m}{N(1-\lambda)} = D_m$$

where $C_m > 1$ is a parameter controlling the threshold and is less than N and is set to 3.

Sparse Grid:

$$D(g,t) \leq \frac{C_l}{N(1-\lambda)} = D_l$$

where $0 < C_l < 1$ and is set to 0.8

Transitional grid:

$$\frac{C_l}{N(1-\lambda)} \leq D(g,t) \leq \frac{C_m}{N(1-\lambda)}$$

Neighbouring grids: Two grids g1 and g2 are neighbours in the k^{th} dimension where $1 \leq k \leq d$, such that:

1. $j_i^1 = j_i^2, i = 1, ...k-1, k+1, ...d$; and
2. $|j_k^1 - j_k^2| = 1$

D-Stream Components

Grid Type conversion A grid can be degraded to from a dense to a transitional or sparse grid or can also be upgraded from sparse to transitional or dense grid depending on whether new data arrives in the grid or not.

The time gap to inspect the density of each grid cannot be either too high or too low as high gap may not detect changes in the data stream properly and low gaps can increase the workload and the amount of computation. Thus, the minimum time needed for the conversion of dense grids to sparse and vice versa needs to be considered and the minimum of these 2 times should be used to set the time gap. This is done in order to make sure that the grid checking is done frequently enough to check the changes in the density of the grid.

Thus, the following results are obtained,

1. Minimum time needed for a dense grid to become a sparse grid,

$$\delta_0 = \left\lfloor log_\lambda \left(\frac{C_l}{C_m} \right) \right\rfloor$$

2. Minimum time needed for a sparse grid to become a dense grid,

$$\delta_1 = \left\lfloor log_\lambda \left(\frac{N - C_m}{N - C_l} \right) \right\rfloor$$

Detection and removal of Sporadic grids Sporadic grids are grids which contains very few data points, this could have happened because of outlier or noise, or even because the structure of the stream changed and the once dense grid has become sporadic with time decay and non-arrival of new points in the grid.

As the number of sporadic grids can become very large considering the case of unlimited data streams with lots of noise and outliers and such sporadic grids can put exceptional load on the system, thus it is important to detect and remove the sporadic grids periodically.

The grids whose density is less than D_l are the candidates for sporadic grids which needs to be considered for removal. There are 2 conditions when the density of a grid can be less than D_l, they are:

1. The grids receive very few data points
2. The grids were once dense but then the density gets reduced as no new points arrive in the system and the decay factors reduce the density of the grid.

The grids in the first case only are considered for removal as they are the true sporadic grids. For this purpose, there need to be a density threshold function which is used to differentiate between these two classes of sparse grids. Thus the density threshold function for a given time t is given as follows,

$$\pi(t_g, t) = \frac{C_l}{N} \sum_{i=0}^{t-t_g} \lambda^i = \frac{C_l(1 - \lambda^{t-t_g+1})}{N(1-\lambda)}$$

where,
t_g = last update time of grid g and $t > t_g$

Given the density threshold function, a sparse grid is adjudged as a sporadic grid if it satisfies the the following 2 conditions. At time t,

1. $D(g, t) < \pi(tg, t)$
2. $t \geq (1 + \beta)t_m$ if grid was deleted before given by time t_m, where β is a constant.

Also, the variables t_m and t_g are part of the characteristic vector of grid g.

After the sporadic grids are detected, they need to be deleted from the grid_list. The grid_list is a list of grids that are considered for the clustering analysis and is implemented as hash tables using doubly linked lists to avoid collision. The key for lookup update and deletion in the hash table is given by the grid coordinates. Thus, as sporadic grids are not to be considered for clustering, thus the following rules are used to delete the sporadic grids from the grid_list:

1. During inspection, all the sporadic grids found by using the previous rules are marked as "Sporadic" but not deleted until the next inspection.
2. In the next inspection, if the grids marked as "Sporadic" have not received any new data points, then the grid g is removed from the grid_list else the density threshold function for g is recalculated and proceeded accordingly.

Once a grid is removed from the grid_list, its density becomes 0 as the characteristic vector of grid g is deleted. This process of deleting sporadic grid helps maintain a moderate number of grids in the grid space S and also saves computing time by not allowing the sporadic grids to grow in numbers. Also, deleting sporadic grids does not affect the final clustering result thus, it is an important part of the D-Stream algorithm.

Clustering Algorithms

Thus the overall process of D-Stream clustering algorithm can be briefly described as follows:

- In the first part, save summary information of every incoming point in the corresponding grids and update the grid densities of all the grids which are part of grid_list. Every dense grid is assigned to a distinct cluster and all the other grids are labelled as NO_CLASS. Also, depending on the density of the neighbouring grids, the cluster labels of one cluster can be changed to the other having more influence.
- In the second part, the grid densities of all the grids which are part of grid_list are updated again. Every grid is analysed whether they are a sparse, dense or transitional grid and based on the observation whether these grids have changed from the last inspection or not, the grids are either inserted or deleted from the grid_list.

Merits and Limitations

Merits

- Clusters of arbitrary shape can be obtained
- Fast clustering by removing non-essential grids
- Able to detect concept drift using decay factor
- No requirement of providing the value of k (Number of clusters)
- Handles outliers effectively

Limitations

- Assumption of high number of empty gris doesn't help in handling high dimensional data well.
- The extent of the grid space S needs to be known beforehand, without which splitting space S is not possible.

3.3.2 D-Stream II

The major part of this algorithm is similar to D-Stream I[15] which is because this is actually an improved version of D-Stream I algorithm.

In D-Stream I, the algorithm simply mapped each data record to a corresponding grid g and computed the grid density. This kind of mapping loses positional information of the data inside the grid. Even though diving the space S in even smaller grids can solve this issue, but then it becomes computationally intensive and thus it may defeat the purpose of clustering streaming data as the input stream may become faster than the computation time for each data point in the grid.

To solve the loss of positional information, D-Stream II[16] considers the attraction of neighbouring grids and integrates this concept with D-Stream I.

Initial Attraction

For a density grid g, let the width at the i^{th} dimension be $2r_i$ and let c_i be the location of the middle point of the i^{th} dimension. Now, for each data point x = (x_1,....x_d) that gets mapped to g, construct a hypercube $Cube(x)$ centered at x whose width is $2\epsilon_i$, where $0 \leq \epsilon_i \leq r_i$ at each of the d dimensions. Now, let $V(x, h)$ be the volume of the intersection of $Cube(x)$ and a grid h, the initial attraction between x and h, given by $attr_{ini}(x, h)$ is given as the ratio of $V(x, h)$ to the volume of $Cube(x)$. That is,

$$attr_{ini}(x, h) = \frac{V(x, h)}{\prod_{i=1}^{d} 2\epsilon_i}$$

Grid Neighbourhood

The neighbourhood of grid g, NB(g) is defined as the set of grids whose center differs from g in at most one dimension.

This makes it easier to define the attraction of a d dimensional data point x = (x_1.....x_d) which is mapped to a grid g, and g' where $g' \in NB(g)$ as

$$attr_{ini}(x, g') = \prod_{i=1}^{d} b_i(x, g')$$

where $b_i(x, g')$ denotes the attraction between x and h in the i_{th} dimension as described in the previous section.

Also, for a d-dimensional data point x = (x_1,....x_d), where x maps to a grid g, then

$$\sum_{h \in NB(g)} attr_{ini}(x, h) = 1$$

Thus the sum of all attraction over all the neighbouring grids comes out to be 1.

Attraction over time

The attraction between x and a grid g at time t, is thus given as follows:

$$attr(x, g, t) = \lambda^{t-t_c} attr_{ini}(x, g)$$

where λ is the decay factor and is exactly the same as defined in D-Stream I.

Also, two grids can have an attraction such that g and h are two neighbouring grids at any given time t. This is known as grid attraction from g to h and is defined as :

$$attr(g, h, t) = \sum_{x \in E(g,t)} attr(x, h, t)$$

where,// E(g,t) is the set of data records that are mapped to g at or before time t.

Also, at this point, it needs to be pointed out clearly that the grid attraction is asymmetric and thus $attr(g, h, t) \neq attr(h, g, t)$ because $attr(g, h, t)$ represents how close the data in g is to h.

Clustering algorithm

As all the other components are same for D-Stream II compared to D-Stream I, we directly jump to the clustering algorithm which needs to take into account the concept of attraction.

In D-Stream II, the attraction between grids are used to decide on the merging of grids. In initial_clustering, the standard density based algorithm merges neighbouring grids to form clusters, although because of the addition of the concept of attraction, two neighbouring grids merge only if they are strongly correlated.

Now, two grids are defined to be strongly correlated if their attractions in both the directions are higher than a threshold $\theta > 0$. Thus, two grids g and h are strongly correlated if $attr(g, h, t) > \theta$ and $attr(h, g, t) > \theta$.

Following the same reason for defining the dense grids, θ can be set as

$$\theta = \frac{c_m}{|\mathcal{P}|(1-\lambda}$$

where $C_m > 1$ and $|\mathcal{P}|$ = number of grid pairs and can be calculated given the number of partitions in each dimension.

The procedure of adjust_clustering updates the density of all active grids for time t where all the active grids are part of the grid_list similar to D-Stream I. Although the clustering procedure slightly differs as in D-Stream II, it uses the information of grid attraction and merges on strongly correlated grids.

Apart from considering the attraction of the nearby grids, the overall algorithms remains the same as in the case with D-Stream I.

Merits and Limitations

Merits

- All the merits of D-Stream I
- Better clusters as the concept of grid attraction is used to specify the clusters better.

Limitations

- All the Limitations of D-Stream I

3.3.3 PKS-Stream

PKS-Stream[17] has a similar initial setting as D-Stream I and II, such that the d-dimensional space S is partitioned into small grids and where every incoming data point x, is assigned a density coefficient and can be calculated at any time t, as

$$d(x,t) = 2^{-\lambda(t-t_0)}$$

where,
λ = the rate of decay also called decay factor
t_0 = the time when the data point arrived in the system

But the similarity with D-Stream ends here as PKS-stream introduces the Pks-tree which mirrors the partitioning of the space S and where each node of the tree corresponds to a grid.

Motivation and Structure of Pks-Tree

The major challenge of clustering high- dimensional data using grid-based algorithm is that it creates a large number of empty grid cells which in general is more that the non-empty grids. In such scenarios, there are 2 methods of proceeding towards clustering the data,

1. All grid cells are stored, which includes the empty ones as well. In this case, the clustering of the data is easy to perform because the relative positional relationships is kept. Though, easier to execute, it needs a large amount of memory to store the empty grid cells.

2. Only the non-empty grid cells are stored, in which case the relative positional relationship is lost. Here, the need for huge amount of memory is

not required as the empty cells are not stored but the process of clustering becomes computationally expensive.

The problems with both the above approach, motivated the creation of the index structure inside the Pks-Tree. It solves the problem by mixing the best of both worlds, thus the empty cells need not be stored anymore and at the same time the relative positional relationship is kept thus improving the efficiency and memory cost.

Structure of Pks-Tree

The definition of the Pks-tree is provided as is given by the authors[17]:
Given the parameter H, and a Pks-tree of rank K ($K > 1$),(a Pks-tree) is defined as follows,

1. The root of Pks-tree at level 1 contains the over synopsis information of space S.
2. Except the root, every node in the Pks-tree is corresponding to a K-cover Grid Cell and stores synopsis information of S at i level (or granularity), which is one-to-one.
3. For any two nodes g_1 and g_2 in the Pks-tree, g_1 is a child of g_2 , if
   - g_1 is a proper sub-cell of g_2 that $g_1 \subset g_2$,and
   - there doesn't exist a node g_3 in the Pks-tree which makes $g_1 \subset g_3$ and $g_3 \subset g_2$.

Also, the height of the tree depends on the value of K such that, more the value of K, lower is the height and the concept of K-cover grid cell helps in removing the empty grid cells so that the overall cost of computation reduces.

Grid Inspection and Density Threshold Function

Similar to D-Streams, PKS-Stream also aims to detect the changes in the data streams, to do this the continuous change in the density of various data grids needs to be inspected. Thus, to detect changes in the grid density in a timely

manner, PKS-Stream calculates the minimum time it takes to convert a dense grid to a sporadic grid. Thus, the minimum time t_{min} is given as follows:

$$t_{min} = \frac{1}{\lambda} log^{\frac{\mu}{\mu-1}}$$

where λ = decay factor and is greater than 0 μ = density threshold and is also greater than 0

PKS-Stream deals with the same problem as D-Stream to find the grids which are really sparse and not the ones which have degenerated from dense to sparse.Therefore, PKS-Stream also employs a Density Threshold Function to detect the true sporadic grids which is defined as follows

$$p(t_c, t_0) = \frac{2^{-\lambda(t_c - t_0 + t_{min})} - 1}{2^{-\lambda t_{min}} - 1}$$

where,
t_c = current time
t_0 = time when grid was inserted into Pks-tree

The value of $p(t_c, t_0)$ keeps on increasing with time thus ensuring that if new data points comes to a sparse grid then they won't be detected as true sporadic grid and thus not being deleted from the tree.

Clustering PKS-Stream

The construction of Pks-tree is imperative for the next step of clustering, thus the Pks-tree building and insertion is done continuously based on the data points and its closeness to a grid cell. Also, the Pks-tree gets adjusted every t_{min} time by using the K-cover grid cell concept. Accordingly,the pks-tree generation algorithm given by the authors is presented in Algorithm 1.

Also, given the optimized Pks-tree, the clustering algorithm is pretty straightforward and is done by clustering all the minimum cells located in the leaf-node level and by assigning neighbouring dense grids with the same cluster label. The clustering procedure of Pks-tree is given in Algorithm 2.

Algorithm 1 Create PKS Tree

procedure PKSTreeCreation($x_1, x_2, ..., x_d$)
  Create root node of the tree, $t_c = 0$
  for current object x^t in stream X **do**
    for `i = 0,1,2,...,H` **do**
      compute the coordinate of the cell g corresponding to x^t ;
      if g exists, then insert x^t into it, goto(4);
      else insert g to the corresponding level of Pks-tree, its $par.childcount++$, and insert x^t into it.
    end for
    $t_c = t_c + 1$;
  end for
  if $t_c \, mod \, per == 0$ **then**
    detect each leaf node of Pks-tree;
    if $g_H.dense < \rho(t_c, t_0)$ **then**
      delete g_H;
    end if
    adjust tree;
  end if
end procedure

Algorithm 2 PKS Clustering

procedure PKSCluster(Pks-tree)
  if $t_c \, mode \, per == 0$ **then**
    for each leaf node grid g_H of Pks-tree **do**
      if g_H is unmarked and g.dense $\geq \mu$ **then**
        mark g_H with a new cluster label;
      end if
      for each neighbour g_H' ofg_H **do**
        if g' 1. meets neighbour g, 2. is unmarked, 3. g_H'.dense $\geq \mu$ **then**
          mark g_H' with the cluster label of g_H';
        end if
      end for
    end for
  end if
  if a clustering request arrives **then**
    generate a cluster
  end if
end procedure

The clusters generated using the minimum cells are updated every t_min time so that the dynamic nature of the stream can be detected and captured in the clusters.

Merits and Limitations

Merits

- Efficient way of dealing with empty grids, improves computational complexity.
- Good for performing clustering on high-dimensional datasets.

Limitations

- Depends heavily on the value of k for k-cover grid cells, this can affect clustering result.
- Difficult to understand the paper with with occasional grammatical mistakes.

3.3.4 DenStream

DenStream[18] is another density based clustering algorithm which unlike the previous density based algorithms reviewed, derives some of the concepts from the partition based clustream and improves upon it.

DenStream overview and Introductory Concepts

Denstream introduces summary representation similar to CluStream [7] which they call core-micro-clusters (or c-micro-cluster) and is defined as a tuple of 3 elements (w,c,r) for a group of similar points each having a timestamp, where, w is the weight, c is the center and r is the radius.

Also, its is imperative to note that a c-micro-cluster is considered dense, only if $w \geq \mu$ and $r \leq \epsilon$. The radius and the weight control the number of c-micro-clusters such that it is greater than the number of natural clusters but lesser than number of data points, which is very similar to what CluStream does, although the number there is user defined.

In addition to c-micro-clusters, DenStream also introduces potential c-micro-clusters (or p-micro-clusters) and outlier-micro-clusters (or o-micro-clusters) which are differentiated based on the weights $w \geq \beta\mu$ and $w < \beta\mu$, respectively.

Also, the entire clustering algorithm employs the damped window model such that the weights contributed by older data diminishes with time and new data gets higher weight. This is given by the following fading formula

$$f(t) = 2_{-\lambda.t}$$

where $\lambda > 0$, is the decay factor

DenStream Clustering

The entire process of clustering the stream is divided into two parts, similar to that of CluStream.

1. Online micro-cluster maintenance - which includes p-micro-clusters and o-micro-clusters as well
2. Offline part of generating macro clusters - which is done based on density unlike CluStream and thus it gives clusters of arbitrary shape.

Online Phase In the online-phase, the c-micro-clusters and p-micro-clusters are maintained for every incoming data points form the stream and a separate buffer called outlier-buffer is maintained for the o-micro-clusters.

Thus, for every incoming point p, p is first attempted to be merged into a p-micro-cluster, else it is attempted to be merged in an o-micro-cluster, both of these operations depend on the radius rp of the point. If neither of the above is possible then a new o-micro-cluster is created and that is inserted into the outlier-buffer.

At the same time, the weights of existing p-micro-clusters keep on decreasing if no new data point arrives in it. Thus similar to D-Stream, a p-micro-cluster needs to be inspected periodically in case it degenerates to an o-micro-cluster and the minimum time needed for this conversion is given by,

$$T_p = \left\lceil \frac{1}{\lambda} log(\frac{\beta\mu}{\beta\mu - 1}) \right\rceil$$

This ensures the maximum number of p-micro-clusters in memory and is given by $\frac{W}{\beta\mu}$

In case there is a lot of noise in the data, the number of o-micro-clusters may increase quickly and keeping them all in the outlier-buffer is expensive, thus DenStream deals with this problem by removing o-micro-clusters whose weight at current time t is lesser than the lower limit of weight which is given as follows,

$$\xi(t_c, t) = \frac{2^{-\lambda(t_c - t + T_p)} - 1}{2^{-\lambda T_p} - 1}$$

where t_o = creation time of o-micro-cluster

The authors then prove mathematically that the total number of micro-clusters increases logarithmically with increasing stream lengths and claim at the same time, that the total number of microclusters in real applications is not going to be very large. The proof is outside the scope of the survey.

Offline Phase In the offline phase, the p-micro-clusters is used to apply a variant of DBSCAN [19] algorithm to get the final macro clusters, where each p-micro-cluster is considered as a point in itself having a weight w.

Similar to original DBSCAN algorithm, the variant creates a macro cluster based on the relative closeness of the p-micro-clusters although this variant includes 2 parameters ϵ and μ such that 2 p-micro-clusters are directly density-reachable if,

1. $dist(c_p, c_q) \leq 2.\epsilon$
2. $w(c_q) > \mu$

For the 2 p-micro-clusters to be density reachable only the first condition needs to be satisfied. At the same time, a sanity check needs to be performed for 2 p-micro-clusters which don't intersect but satisfy the 1st condition as the actual radius is smaller than epsilon. To detect such a case, a sanity check is performed as follows

$$dist(c_p, c_q) \leq r_p + r_q$$

where r_p and r_q are radius of c_p and c_q respectively.

Merits and Limitations

Merits

- Well defined strategy to differentiate between outliers and potential micro-clusters.
- Very useful for dynamic data streams in detecting concept drifts.
- DBSCAN can result in decent clusters of arbitrary shape.

Limitations

- New micro-clusters for every incoming data point, if it can't be merged into existing micro-clusters.
- Even with regular pruning, number of o-microclusters can increase a lot and exceed memory constraints.

3.4 Model Based

Model Based Clustering: Model-based clustering methods attempt to optimize the fit between the given data and some mathematical model like EM (Expectation Maximization) algorithm. The following algorithms attempts to do model based clustering on streaming data

3.4.1 SWEM

The authors of SWEM[20] define it as "clustering data streams in a time-based Sliding Window with Expectation Maximization technique". It considers the task of clustering the data stream in a sliding window such that older data can be eliminated in a structured way.

SWEM Structure

SWEM's structure focuses on a sliding-window model which is time-based.

The entire time horizon is divided into smaller time periods as $TS_0, TS_1 ... TS_i$, where in any given time period any number of d-dimensional data points x =$\{x_1, x_2, ...x_d\}$ could arrive.

Now the set of data points which arrived over the last b time periods can be defined by a time-based sliding window $SW = \{TS_{i-b+1} .... TS_{i-1}, TS_i\}$ where, TS_i = latest timeslot, TS_{i-b} = Expired timeslot

Now, the authors assume that the data points are generated using some dynamic statistical process, which helps evolve the stream with time and that the resulting k clusters follow a multivariate normal distribution. Based on the above assumptions, any cluster C_h, where $1 \leq h \leq k$ has a characteristics parameter ϕ_h given as

$$\phi_h = \left\{\alpha_h, \mu_h, \sum\nolimits_h\right\}$$

where,
α_h = cluster weight,
μ_h = vector mean,
σ_h = covariance matrix

Thus, SWEM aims to find the set of k parameters $\Phi_G = \{\phi_1, .., \phi_k\}$, which fits the data that arrived in the last b time slots.

SWEM Clustering algorithm

The entire process of SWEM is divided into 3 phases, these phases are described briefly.

Initial Phase In this phase, m distributions or micro components are created which models the data over the b timeslots in the sliding window. The set of m parameters of the m micro components is given as $\Phi_l = \{\phi_1 .... \phi_m\}$. Also, for the initial phase the initial value of the parameter are chosen at random.

It is also assumed that the generation of data points is independent and the average log likelihood of n records in TS_0 is given as follows,

$$Q(\Phi_L) = |TS_0|^{-1} log \prod_{x \in TS_0} \sum_{h=1}^{m} \alpha_l \times p_l(x_i|\phi_l)$$

Thus SWEM uses EM method to maximize the value of $Q(\Phi_L)$ and thus when the algorithm finally converges, for each MC_l the above set of calculated micro components are approximated as a triple T_l as

$$T_l = \{N_l = |S_l|, \theta_l = \sum_{x_i \in S_l} x_i, \Gamma_l = \sum_{x_i \in S_l} x_i x_i^T\}$$

T_l helps in computing the mean and covariance of MC_l and at the same time the additive property of T_l ensures that when components are merged, the new T_l of the merged component can be easily calculated.

Having calculated all the sufficient statistics, the k clusters are computed using expectation maximization as follows:

E-step:

$$p(\phi_h|T_l) = \frac{\alpha_h^{(t)} \times p_h(\frac{1}{N_l}\theta_l|\mu_h^{(t)}, \sum_h^{(t)})}{\sum_{i=1}^{k} \alpha_i^{(t)} \times p_i(\frac{1}{N_l}\theta_l|\mu_i^{(t)}, \sum_i^{(t)})}$$

M Step:

$$\alpha_h^{(t+1)} = \frac{1}{n}\sum_{l=1}^{m} N_l \times p(\phi_h|T_l); \qquad \mu_h^{t+1} = \frac{1}{n_h}\sum_{l=1}^{m} p(\phi_h|T_l) \times \theta_l;$$

$$\sum_h^{(t+1)} = \frac{1}{n_h}\left[\sum_{l=1}^{m} p(\phi_h|T_l)\Gamma_l - \frac{1}{n_h}\sum_{l=1}^{m}(p(\phi_h|T_l)\theta_l)(p(\phi_h|T_l)\theta_l)^T\right]$$

where $n_h = \sum_{l=1}^{m} N_l \times p(\phi_h|T_l)$

Incremental Phase In this phase, the converged parameters in the last phase are utilized to get the starting list of values for the parameters in the mixture model.

Also, this phase deals with merging and splitting of components when there is considerable changes in the streams distribution. Thus, smaller components are

merged, if they are close together and bigger components are split if they are large and have highest variance sum.

Expiring Phase As time moves, the sliding window slides as well, and thus the oldest time slot is eliminated from consideration. The list of new k parameters Φ_G is updated by subtracting the summarized statistics from the expiring timeslot. SWEM helps in gradual removal of such statistics by employing a decay factor λ which reduces the weight of the expiring MC_l by a factor of $\lambda^t N_l$ after each iteration, where N_l = Number of data points assigned to S_l having high probability.

Merits and Limitations

Merits

- Strictly single scan
- EM technique used has solid mathematical application

Limitations

- The mixture model is assumed to follow Gaussian distribution
- Paper structure not very clear.

3.5 Algorithm Comparisons

In this section, we look through all the algorithms at a glance and present a simple yet comprehensive comparison of the algorithms reviewed. This comparison is by no means exhaustive, although for the reader its useful to understand the features offered by every algorithm. The comparison is presented in a tabular form in table 3.1,

Stream Clustering Algorithm Comparison				
Algorithm	Year	Algorithm Type	Cluster Shape	Concept Drift Detection
STREAM	2003	Partition-Based	Spherical	No
CluStream	2003	Partition-Based	Spherical	Yes
StreamKM++	2012	Partition-Based	Spherical	No
BIRCH	1997	Hierarchical-Based	Spherical	No
ClusTree	2011	Hierarchical-Based	Depends*	Yes
ODAC	2006	Hierarchical-Based	Spherical	Yes
D-Stream I	2007	Density-Grid-Based	Arbitrary	Yes
D-Stream II	2009	Density-Grid-Based	Arbitrary	Yes
PKS-Stream	2011	Density-Grid-Based	Arbitrary	Yes
DenStream	2006	Density-Grid-Based	Arbitrary	Yes
SWEM	2009	Model-Based (EM)	Spherical	Yes

TABLE 3.1: Comparison of reviewed algorithms

*ClusTree's macro clustering step depends on a traditional multi-pass clustering algorithm. Thus, if an algorithm similar to K-Means is chosen for clustering then spherical clusters are generated, else if density based clustering is applied then the macro clusters generated would be of arbitrary shape.

Chapter 4

Algorithm Selection and Implementation

In this section, we discuss the algorithm which was selected for implementation from the pool of surveyed algorithms. Along with that, we will also discuss the reasons behind selecting the algorithm for implementation and discuss the pro's and con's of the same.

Later, details of the implementation of the selected algorithm is presented along with brief introduction to the scalable data-processing engine-Apache Flink.

4.1 Algorithm selected for Implementation

After thorough understanding of all the surveyed algorithm, we decided to select CluStream for implementation purpose from the pool of the 11 algorithms which were surveyed in the chapter 3.

4.2 Reasons for selection

The main reason of selecting CluStream for implementation, is the novel concept of online - offline framework which suits the streaming data scenario particularly well. As streaming data consists of continuous, fast and potentially unlimited data, thus a clustering algorithm over such a streaming data should be able to

quickly process such data points. Also the data points are generally one pass in nature, such that once it is processed it is gone. Thus, the online component which creates microclusters introduced by CluStream helps process the data quickly and keep a summary of all the incoming data points, such that macro clusters can be later in the offline phase created as per the wish of the end user.

Another very important reason of selecting CluStream is the fact that a very large proportion of the stream clustering algorithms mimic the concept of microclusters and online-offline framework, even with different methodologies. Thus, an implementation of CluStream could be extended for the other similar algorithms with minimal effort.

Obviously, there are algorithms which perform better and give better clusters. But implementing a stream based algorithm on a large scale & parallel architecture has its set of challenges. Therefore, the selection of CluStream seemed optimal for current use and for future research as well.

A working implementation of CluStream over the parallel architecture would open up new possibilities of implementation and usage of stream based algorithm on a large scale, thus a working model of an algorithm is more important than an algorithm which might present better results but is non-functional.

We now discuss about the implementation details with a brief introduction to the scalable stream processing engine - Apache Flink, followed by the workflow and challenges associated with implementing the clustering algorithm on top of Flink.

4.3 Flink Introduction

Flink[21] is a high performance scalable batch and stream processing engine which can process really large amount of data with ease. Here, we are more concerned about stream processing, so our discussion would be confined to the boundaries of handling streaming data.

Flink's core is a streaming dataflow engine that provides data distribution, communication, and fault tolerance for distributed computations over data streams. It is a result of of the Stratosphere[22] project whose primary objective was to "enable the extraction, analysis, and integration of heterogeneous data sets, ranging from

strictly structured relational data to unstructured text data and semi-structured data."

Flink's [23] dataflow engine is written in Java and Scala and it can execute a given dataflow program in a distributed, data parallel and pipelined manner. Flink programs can be written using high level languages like Java and Scala. The Flink community is also working on building a python API which would enable programmers to run Flink jobs using Python as well.

Flink can seamlessly connect with various data sources and even with other open source technologies, for e.g.[21] it runs on Yarn, integrates with HDFS for data storage, consuming data streams from Kafka like Apache Kafka, for streaming data, consumption Flink runs on YARN, works with HDFS, streams data from Kafka, and can even execute Hadoop program code.

4.3.1 Flink Features

Flink boasts of high throughput and low latency stream processing with exactly-once guarantees. Some of the features offered by Flink[21] are described briefly:

- **High Performance & Low Latency** - Flink's data streaming runtime achieves high throughput rates and low latency with little configuration.
- **Support for Event Time and Out-of-Order Events** - Flink supports stream processing and windowing with Event Time semantics.Event time makes it easy to compute over streams where events arrive out of order, and where events may arrive delayed.
- **Exactly-once Semantics for Stateful Computations**- Streaming applications can maintain custom state during their computation.Flink's checkpointing mechanism ensures exactly once semantics for the state in the presence of failures.
- **Highly flexible Streaming Windows**- Flink supports windows over time, count, or sessions, as well as data-driven windows.Windows can be customized with flexible triggering conditions, to support sophisticated streaming patterns. Thus one can maintain a window of events to emulate batch behaviour or perform calculations which are not directly possible with continuous stream processing.

- **Continuous Streaming Model with Backpressure**- Data streaming applications are executed with continuous (long lived) operators.Flink's streaming runtime has natural flow control: Slow data sinks backpressure faster sources and it can handle such scenarios well.
- **One Runtime for Streaming and Batch Processing**- Flink uses one common runtime for data streaming applications and batch processing applications.Batch processing applications run efficiently as special cases of stream processing applications. This benefits Flink greatly as it doesn't have to compromise on stream processing and at the same time emulate batch behaviour as well.

4.4 Implementation Details

In this section, we would describe the workflow of our implementation. This would include details about reading the data, preprocessing it if necessary, creating microclusters and then correspondingly creating the macro clusters. But before going into the details, we looked into the non-parallel, non-scalable version of CluStream from [24] which gave insights as to how CluStream worked.

The process starts by reading the datapoints either from a datastream source or creating a wrapper around a datafile such that data points are streamed from the file. This process parses the data and if required, pre-processes it by removing redundant attributes or marking labels if available, as well.

Once the datastream generator starts generating data points, it is fed to the Flink iteration which according to the Flink iterate javadocs[25], "initiates an iterative part of the program that feeds back data streams". The data stream which is fed back is the updated set of microclusters which are created after the datapoints have been consumed in the iteration process. This process is the trickiest part of the entire workflow and it includes a lot of challenges which are described in the sections [4.5, 4.6]. Once the microclusters are created, it is fed back for the next iteration and at the same time a copy of the microclusters are forwarded for storage, such that macro clusters can be created from them.

For creation of the macrocluster, we used an implementation of KMeans++ seeding procedure similar to [26], thus the initial centroids are found using this method.

Once we have the centroids, we just treat the microclusters as data points and create macroclusters based on naive implementation of Kmeans using the centroids already found.

Once we have the macro cluster several performance metric like SSE and purity of clusters are calculated. In our implementation, we keep the data points temporarily even after their consumption in the microcluster updation process. This is done so as to analyse the purity of the cluster as these data points are tested against the calculated labels and original labels.

```
ConnectedIterativeStreams<Point, MicroCluster[]> inputsAndMicroCluster =
                                        tuples.iterate(1000)
                                        .withFeedbackType(MicroCluster[].class);

DataStream<Tuple2<MicroCluster[],
                List<PointLabel>>>
                updatedMicroClusterWithPoints =
                                inputsAndMicroCluster
                                .flatMap(new MyCoFlatmap(num_of_mc,tw, startTime))
                                .assignTimestampsAndWatermarks(new
                                BoundedOutOfOrdernessGenerator())
                                .keyBy(new KeySelector<Tuple7<MicroCluster[],
                                                Integer,Long,Point,Integer,Long,Long>,
                                                Long>() {

                                        @Override
                                        public Long getKey(Tuple7<MicroCluster[],
                                                        Integer, Long, Point, Integer,
                                                         Long, Long> value)
                                                        throws Exception {
                                                return value.f6;
                                        }
                                })
                                .countWindow(totalParallelism)
                                .fold(new Tuple4<MicroCluster[], Integer,
                                                List<PointLabel>, Boolean>
                                                (initialMC, -1, initialPointLabel,true),
                                                 new FoldMC())
                                .flatMap(new ReturnMC());

DataStream<MicroCluster[]> updatedMicroCluster =
                                                updatedMicroClusterWithPoints.
                                                flatMap(new GetMC());
inputsAndMicroCluster.closeWith(updatedMicroCluster.broadcast());
```

Code 4.1: Iteration code snippet

The code snippet presented in 4.1, presents the main iterative logic of our implementation, the iteration starts from the *tuple.iterate*(10000) call where *tuple* is our

data source stream and 10000 is the maxWaitTime parameter to set a max waiting time for the iteration head. If no data is received in the set time, the stream terminates. After calling our flatMap function, the flatmap receives data from both the data stream source and the iterative feedback sources, if data is available. We then process the data points and create corresponding microclusters which are then sent back to the stream element $updatedMicroClusterWithPoints$.

The $updatedMicroClusterWithPoints$ in our code helps us in initiating the iterative feedback part of the program. The iterative part needs to be closed by calling the $connectedIterativeStream.closeWith(DataStream)$ which is done in the last line of the code snippet. This data stream which is given to the $IterativeStream.closeWith(DataStream)$ method is the data stream that will be fed back and used as the input for the iteration head. The user can also use different feedback type than the input of the iteration and then specify the feedback type in the $TypeInformation$ field of $IterativeStream.withFeedbackType$ $(TypeInformation)$. A common usage pattern for streaming iterations is to use output splitting to send a part of the closing data stream to the head and forwarding the other part based on some filter specified.

A visual representation of the workflow from the Flink Dashboard is shown in 4.1:

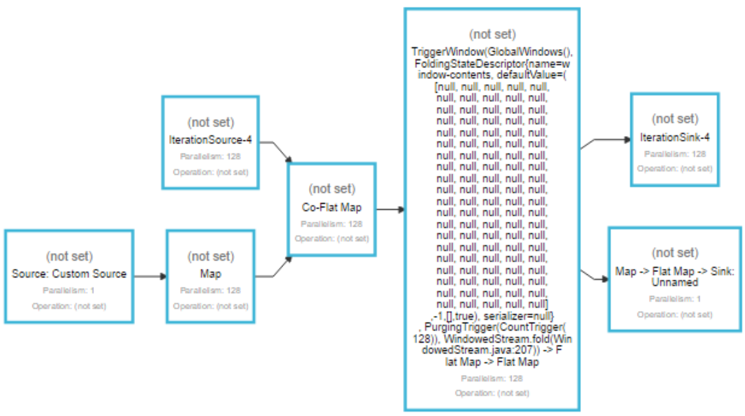

FIGURE 4.1: Original Workflow from Flink Dashboard

Once the microclusters are obtained from the map operation, the microclusters are then fed to the macro clustering procedure which creates the given number of macro clusters.

4.5 Flink Challenges and Limitations

During the course of our implementation of CluStream we had to go through a lot of challenges, this is primarily because of the reason that, even though Flink is great for handling and processing of large amounts of streaming data but at the same time it is not completely mature in terms of solving machine learning problems and there still exists a lot of limitations.

Also, all the surveyed algorithms claim that they work on large scale and distributed platform and conceptually they might be correct but none of the algorithms were tested on such environment. All the algorithms which we surveyed during the course of this project were implemented and tested on single machines with limited storage and memory. The distributed environment brings in additional features like more computing power and more memory but at the same time it introduces more challenges. Such challenges include proper distribution of the datasets to different partitions, processing of the data in each of the individual partitions and then devising a way to combine all the data together to obtain correct results.

In addition to the above, there were some functionalities which were available for the static data via the DataSet API[27] but the same wasn't available in the DataStream API[28] for the unbounded stream of data, for e.g. unavailability of broadcast sets to broadcast the list of centroids. This and other limitations are described in details in the following section which discusses the Implementation challenges.

4.6 Implementation Challenges

As we already discussed the challenges imposed by the underlying architecture and also the fact that the algorithms were not tested on distributed environments, now is the time to discuss the challenges we went through while implementing our selected algorithm - CluStream. Thus, we list down the number of challenges faced and what was done to tackle them in this section:

1. Unavailability of Broadcast variables This is one of those features which is available in the DataSet API but missing in the DataStream API, and its very

crucial as Broadcast variables allows one to make a data set available to all parallel instances of an operation, in addition to the regular input of the operation. This is useful for auxiliary data sets, or data-dependent parameterization. The data set will then be accessible at the operator as a Collection.

Thus, one can easily broadcast the list of centroids or like in our case,set of microclusters to all the partitions for every iteration. Also, this allows one to integrate the list of microclusters for initial n data points which are already calculated beforehand as this is one of the requirements of the CluStream algorithm.

These things could only be done with the DataSet API, this is because this feature is still under development for the DataStream API and is not available for use.Therefore, our algorithm had to improvise and take into account the unavailability of broadcast variables for use in the Streaming context.

To solve this issue of unavailability of broadcast variables, we tried different alternatives and tested out different configurations with our implementation. Also at this time, it is important to note that, there is broadcast available for the Streaming Context, but its not similar to the broadcast variables in the Dataset API and this function just broadcasts the elements to all the partition. The usage of this broadcast functionality is non-trivial and is bounded with a datastream variable. Although, as broadcasting of the set of microclusters is an integral part of our clustering algorithm, we had to make use of what was available to us.So, for broadcasting the microclusters following approaches were tries:

1. We read n data points and created initial list of microclusters, but as it was not possible to send this microcluster directly into all the partitions and access it (which is evident from code snippet 4.1), thus we tried storing/discarding the list of data points until the list of microclusters is broadcasted back via the iterator to the datastream variable which is connected to the main data stream.

   This alternative meant that either we keep on storing the incoming data points which risked overflow of the operator or we discard data which meant that a lot of data points are ignored. Thus this alternative may seem to work but its not the best one to follow in practice.

2. We don't calculate the initial list of microclusters in the beginning and we calculate them on the fly. This meant deviating from the exact CluStream

model a bit but in a parallel distributed environment which already has so many restrictions, such deviation, if beneficial, can be done.

Thus, we get the data points in each of the partitions and then calculate diffierent initial microclusters. This behaviours is problematic as the initial microclusters are different thus we can't reduce the microclusters based on the microcluster id as it would give very skewed result. But on the longer run we assumed that the clusters would stabilise as more data points come in. Although during our experimentations we found out that rather than fixing of the clusters , the clusters got more skewed and the centers moved away from the actual cluster centres for atleast one of the clusters.

2. Reducing the microclusters from partitions As seen already, the reducing procedure of microclusters is a problem if we choose to not have the initial set of microclusters which has its own drawbacks. But along with the accuracy issues, there is an additional problem of accumulating the broadcasted microclusters. This is explained as follows:

When a microcluster is broadcasted, its copies are sent to each of the indidvidual partitions. In each of this individual partitions the data points are consumed as they arrive and according to the CluStream algorithm the summary information of each of the data points is added onto the microcluster which it closely resembles to, upto this part the procedure is fine and each of the partitions perform the assigned task properly. But after this, the microclusters need to be aggregated via the reduce procedure, now the problem here is that as each of the paritions had an initial set of microclusters. Thus, in the reduce procedure, we get n copies of the initial microcluster in addition to the consumed data points from each of the partition. Dealing with this issue was very essential to not get outrageous microclusters.

4.7 Design details

We designed the implementation in such a way so as to deal with the challenges presented in sections 4.6 & 4.5. Our design helps solve both of the problems described in section 4.6 at the same time.

What we decided to do is to generate the initial set of microclusters on the fly in the individual partitions. To solve the problem of skewed microclusters, we dont merge microclusters from different partitions based on the microcluster id. Rather, in the reduce phase we merge 2 set of microclusters based on the distance of one microcluster from one reduce set to all the microcluster on the other reduce set. The merging process is very similar to assigning data points to corresponding clusters using K-Means. This procedure fixes the problem of skewed microclusters and thus we have proper microclusters after every iterations.

Now to solve the problem of n copies of the initial microcluster, we decided to simplify the microclusters to their cluster centres by dividing the aggregated data points with the total number of points and then merging incoming data points to the normalized microclusters. This is not optimum as the density information of each microcluster is lost but given the scenario, this is the best possibility we could come up to after brain storming.

These steps solved our implementation challenges. Although we had to introduce some design changes to the original clustream algorithm, but it was necessary as it helped achieve better and accurate cluster centers which is the main objective of any clustering algorithm.

4.8 Original Work

CluStream performs clustering over the data and thus we only have the updated cluster centres so as to understand where the cluster is evolving with time. This behaviour is optimum for clustering algorithms, but as we know that the clusters produced by such algorithms can be used for classification purpose, so we decided to extend the classic CluStream model to perform classification as well. To do this, we need the original data points to classify but after creation of microclusters, CluStream consumes them and they are not used anymore.

Thus, we extended our model to perform classification of incoming data points based on the current set of macroclusters which were created from microclusters. We do this by not discarding the incoming data points after being consumed in the microcluster but keeping it for some more time and accumulating a list of points from the different partitions and then classifying the clusters associated with the data points.

This increases the running time a bit, but as the list of points are only clustered and labelled for the current time t, thus the overall time doesn't increase by a huge factor.

Even though this part of the implementation is useful, still this concept of identifying the clusters from the incoming points introduces 2 different problems,

1. We don't have the points for recalculation after the cluster moves. This is not a problem as we define the problem of clustering based on the current incoming points and these points don't play a a major role in the future.

2. We get the labels for the incoming points thus, the points which are helping create the macro clusters are being labelled by the microclusters. This in a way is not correct in terms of classification but in clustering a data point can be marked to the label of the cluster it has been assigned to. Thus, this can be done in our case to represent points in their home clusters.

Finally, after mentioned changes/improvements, we have our own version of CluStream.

Chapter 5

Evaluation

This chapter refers to all the experimentations performed over different platforms. We have a lot of parameters to test our implemented algorithm. Therefore, we found the best set of parameters using entire grid search. This however, didn't perform well as older jobs started taking more time than usual which in turn produced skewed output.

Due to the aforementioned reasons, we tested on specific sets of parameters to show how well our implementation of CluStream works.

5.1 Evaluation Metrics

For evaluation of our algorithm, we choose the following 3 metrics:

1. Latency - Latency for our algorithm is calculated by taking the total time to process n data points and then dividing it by n to get the average time for processing a single data point completely.
2. Purity/Accuracy of the Clusters - As we generate the data ourselves, thus we check the accuracy by comparing the cluster assigned by our algorithm against the original label for each data point. Thus, the final accuracy is given by

$$Accuracy = \frac{CorrectlyIdentifiedPointsinaCluster}{TotalPointsintheCluster}$$

3. SSE (Sum of Squared Errors) - SSE gives us the compactness of the cluster created by the algorithm compared to the original clusters. Thus, we compare the original SSE against the calculated SSE and define our cluster compactness, which again means that the clusters are well defined.

5.2 Comparison

For comparison purpose, we decided to compare our implementation against Apache Samoa, Apache Spark and MOA. Although we could only perform our comparison against MOA. This is because of the following setbacks:

- We tried running Apache Samoa on top of Apache Flink using the connector available, but we couldn't make it work. For this, we reached out to the developer group for Samoa and we got a response that the support for Apache Samoa is low and for the connector on Flink its even lower and full of bugs. Thus, we dropped Apache Samoa.
- We were not of the opinion to compare our implementation against Apache Spark's own implementation of stream clustering algorithm - "Streaming Kmeans". This was because, the spark implementation of Streaming Kmeans relied on 2 different data stream, one for training and one for testing. Normally in streaming data, we have only one source, and even if we have multiple data sources, we still don't have a train and a test stream. Also, the algorithm was not capable of adapting the model based on the test stream, thus defeating the purpose of data stream clustering. We still tried running the spark implementation, but even after multiple correspondence from the Spark community, our problem was not solved. Thus we decided to leave Spark out as well from comparison.

MOA Framework

MOA[29] is an open source data stream mining framework which works in a non-distributed environment.It includes a good collection of various machine learning algorithms which not only includes common methods like classification, regression, clustering but also uncommon but useful methods like outlier detection, concept

drift detection and recommender systems. It also features tools for evaluating the output of the machine learning task. It is written in Java and has close association to Weka[30] which is a popular collection of machine learning algorithms for data mining tasks.

MOA provides both, a graphical user interface(GUI) and a command line interface(CUI). Although we directly use the java code as it was more accurate with our comparison.

One of the reason, we wanted to compare our scalable implementation to the non-scalable version is because we wanted to show that the performance our system is better or comparable in a scalable environment.

MOA works on .arff data format which is inherited from Weka. Thus, all the data sets we generated had to be converted to .arff format before working on them.

5.3 Experiments and Results

For experimentation purpose, we generated 2 sets of dataset, one for testing on the cluster and one for testing on the local machine. The data is generated using multivariate dataset generator, which was created in Java. For the purpose of proper experimentation on the cluster, we generated dataset of sizes [100MB, 512MB, 1024MB], also for each of the dataset size we generate data with different numbers of clusters [3,5,10] and dimensions [10,50,100]. And to test them on local machine, we generated dataset of sizes [10,50,100], with other parameters remaining the same.

In addition to these parameters, we tested our implementation on the cluster for different parallelism [32,64,128] and for different number of microclusters(which according to CluStream[7] should ideally be 10 times the number of clusters), and so chose the following range of values for number of micorclusters [30,50,100].

All the data we generated had 10% noise within it, so as to make the data closer to real datasets and also to make the clustering process more challenging.

Tests on the Cluster

We would first present the result of our implementation on the cluster. The cluster available to us have one master node and 8 task managers/slave nodes making a total of 9 nodes, each having 48 cores and 62.1 GB of physical memory out of which Flink manages around 20.4 GB for operations. We used Apache Flink version 1.0.0 for testing our implementation.

For all the results published, we tested 3 times for the same set of parameters and took an average of the same. This was done so as to take into account any irregular behaviour within the system.

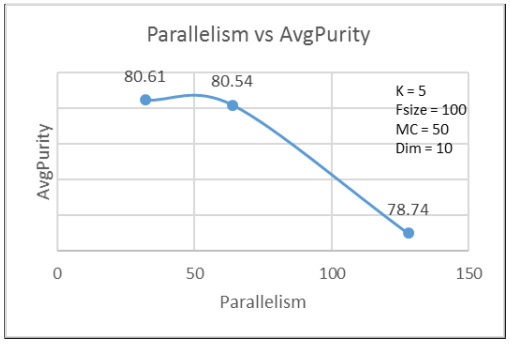

FIGURE 5.1: Avg-Purity for Different Parallelism

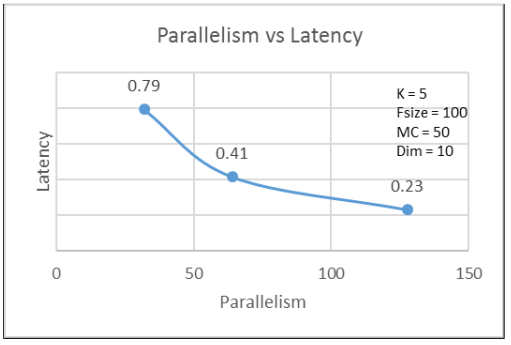

FIGURE 5.2: Latency for Different Parallelism

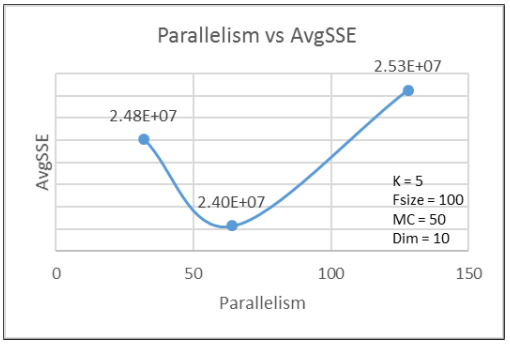

FIGURE 5.3: Avg-SSE for Different Parallelism

Figures 5.1, 5.2, 5.3, as mentioned refers to the comparison of different amount of parallelism against the metrics defined in section 5.1 keeping other variables constant. As can be seen from the charts, with increase in parallelism, the latency of our implementation decreases which is a good sign as the aim of scalable design was to reduce latency. But at the same time the purity of the clusters go down for increasing parallelism, given the fact that more number of microclusters needs to be maintained across more partitions and the macro clustering phase has more

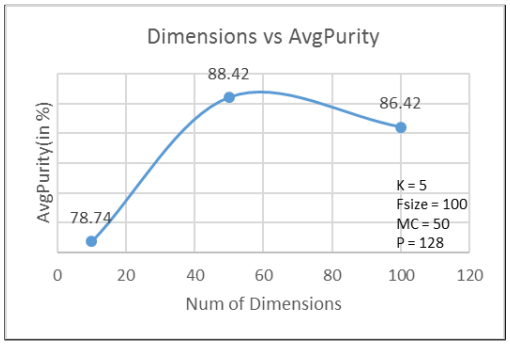

FIGURE 5.4: Avg-Purity for Different Dimensions

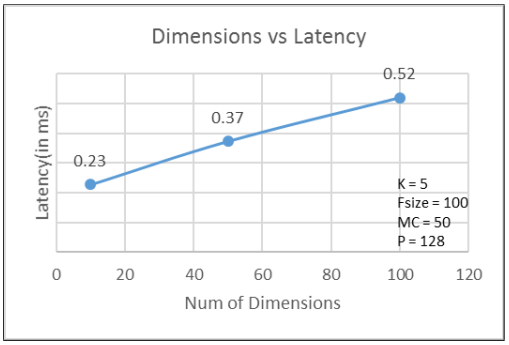

FIGURE 5.5: Latency for Different Dimensions

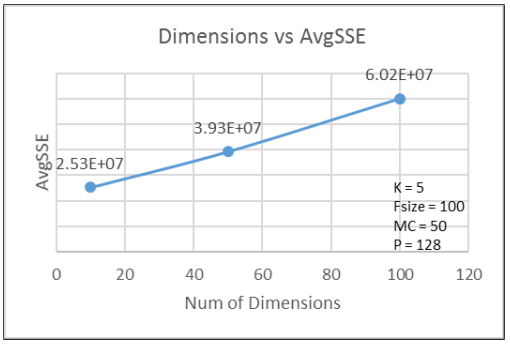

FIGURE 5.6: Avg-SSE for Different Dimensions

number of points to choose for centroid selection. Also, the AvgSSE increases given the fact that the cluster purity goes down.

Next, in figures 5.4, 5.5, 5.6 we compare the metrics against the dimensionality of the data. As expected, the latency and avgSSE increases with increase in the number of dimensions as more calculation is needed to be done for distance and more attributes contribute to the SSE of each cluster respectively. The purity increases and then slightly decreases a bit, as dense cluster across dimension can be both easy and difficult to create.

In the next set of figures 5.7, 5.8, 5.9, we keep all the other parameters fixed as shown in the chart except the number of microclusters. This result is specifically interesting for observation as usually the number of clusters is chosen by user. As we can see from the charts, the avgPurity increases as we increase the number of clusters from 30 to 100 but at the same time the latency goes up as well. Thus , an optimum value for the number of microclusters needs to be selected such that the latency is not high and the cluster purity is obtained as desired. Also, avgSSE decreases in this case given the increase in cluster Purity.

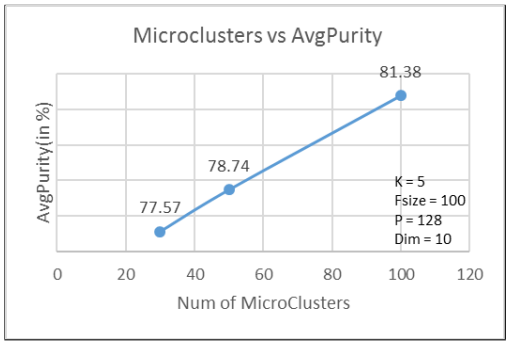

FIGURE 5.7: Avg-Purity for Different MicroCluster

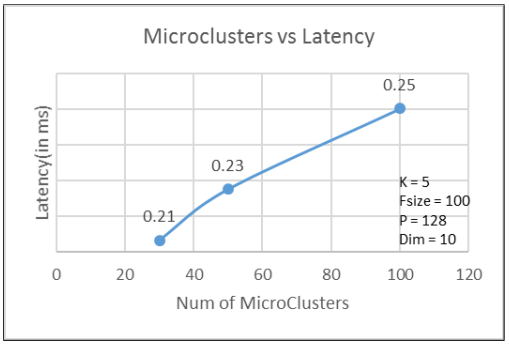

FIGURE 5.8: Latency for Different MicroCluster

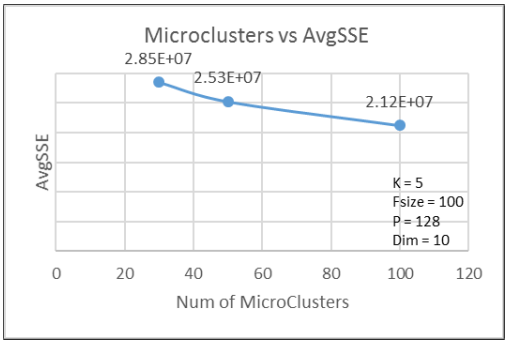

FIGURE 5.9: Avg-SSE for Different MicroCluster

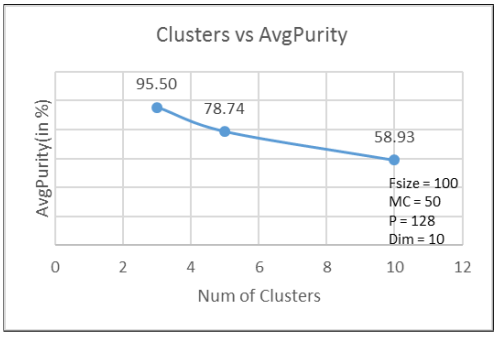

FIGURE 5.10: Avg-Purity for Different Cluster

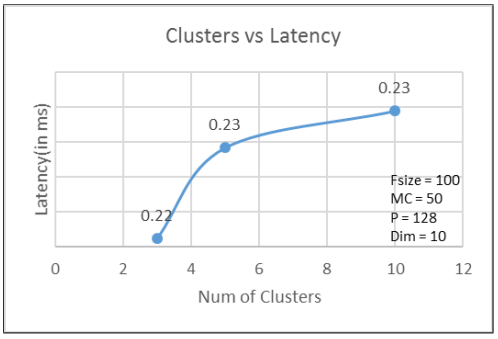

FIGURE 5.11: Latency for Different Cluster

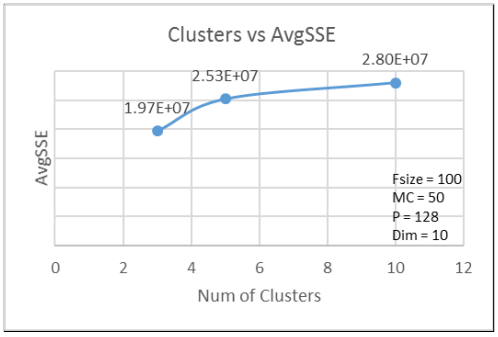

FIGURE 5.12: Avg-SSE for Different Cluster

Now we see the behaviour of the metrics in terms of number of clusters as seen from figures 5.10, 5.11, 5.12. This is the worst performing aspect of our implementation, as the latency increases as we need to do more calculations and the cluster purity is affected heavily. The reason for this could be the limited number of points to select the cluster centroids. Thus, if the number of microclusters are increased according to the number of clusters, we won't see such drastic reduction in the cluster purity.

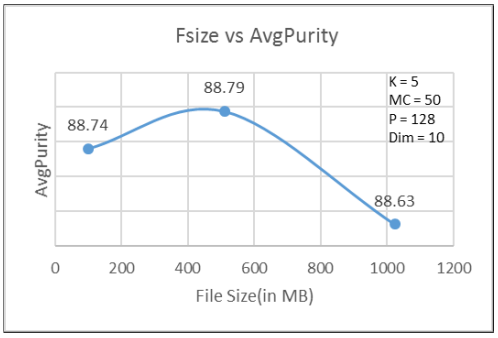

FIGURE 5.13: Avg-Purity for Different FileSize

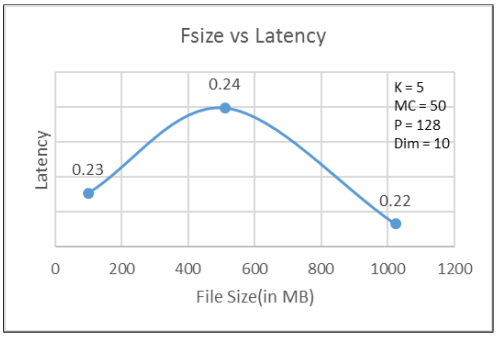

FIGURE 5.14: Latency for Different FileSize

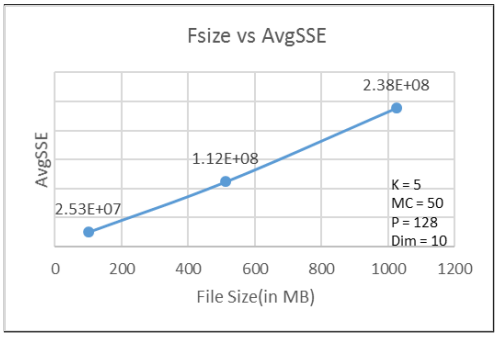

FIGURE 5.15: Avg-SSE for Different FileSize

The last set of test done on the cluster is for different values of file sizes ranging from 100 MB data source to 1 GB, which are shown in figures 5.13, 5.14, 5.15. Even with multiple tests, the latency across the different file sizes increases a bit then goes down for files of higher sizes. The behaviour is same across the purity but as we can see, the purity is not heavily affected. This shows that we can have higher amount of data without adversely affecting the system.

Now, we take a look as how much the AvgSSE is affected after clustering, as mentioned in section 5.1, SSE gives us the compactness of the cluster. So, when the cluster purity goes down or the number of dimensions increase, the SSE should also increase. In our case we compare the SSE present in the original data against the SSE which we obtained from our system and try to find trends in it.

Figures 5.16, 5.17, 5.18 gives the trend of the AvgSSE of our system against that of the original data for different filesizes, number of clusters and number of dimensions respectively. As expected, the AvgSSE increases in each of the cases, but the compactness increases by a bigger factor than compared to the data.

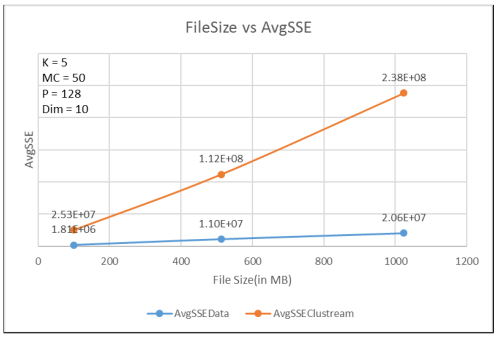

FIGURE 5.16: Avg-SSE for Different FileSize

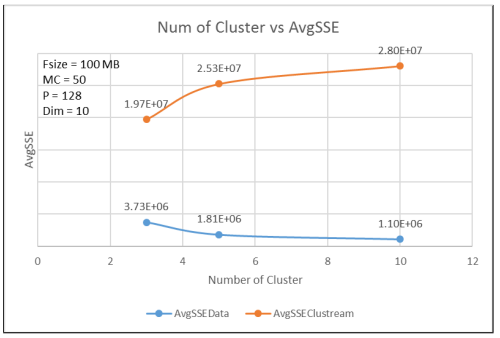

FIGURE 5.17: Avg-SSE for Different Number of Clusters

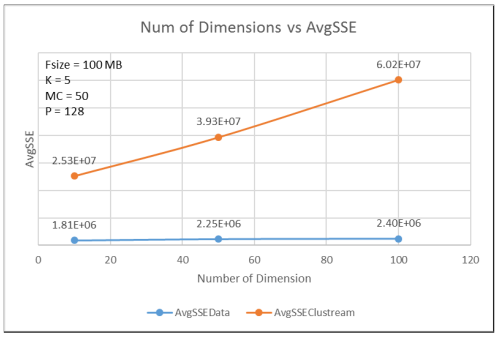

FIGURE 5.18: Avg-SSE for Different Number of Dimensions

In our current scenario, the AvgSSE of our system is generally between 5 to 25 times the value of AvgSSE from the original data. This needs to be taken care of and the only way of reducing the AvgSSE is by improving the clustering process further, to create better clusters with higher purity.

Tests on local machine

We also tested our implementation on local machine with 2 cores(4 logical processors) having 8 GB physical memory. We ran our implementation on top of Apache Flink 1.0.0 which is the same version on the cluster.

The results obtained on the local machine follow more or less the same trend as on the cluster and thus are not individually presented here.

Comparison

In this section, we present the results of comparison of our system on cluster and on local machine against MOA [5.2]. MOA doesn't give us the output of the clustering process, thus we can't compare the avgSSE of the system with the results obtained from our system.

It's important to note at this point that MOA is a highly optimized product with lots of user and the result it provides can be better and faster due to this optimization. Also, in MOA, the number of microclusters is fixed at 100, so we didn't change it and let it be the default setting. Also,we couldn't get the purity of the cluster from the Java code, but when we ran the implementation on the given the highly optimized nature of MOA, the purity of the of the clusters obtained

We first compare as to how the number of dimensions affect latency and purity on Flink cluster, Flink local and MOA. The figures 5.19, 5.20 illustrates the results. From the chart, it is clear that even though our implementation of CluStream on local machine doesn't perform well, it fared better than MOA for higher dimensions in terms of latency. This result is important as data streams could consist of data having high number of attributes and even though CluStream is not the best when it comes to handling high dimensional data, still it performs well.

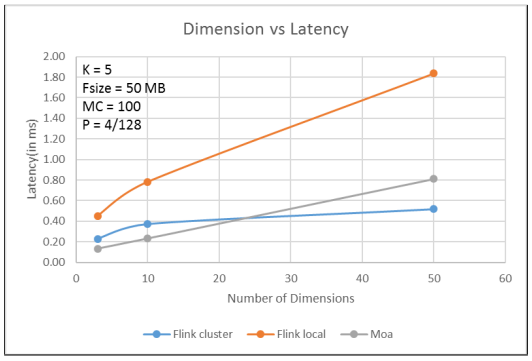

FIGURE 5.19: Latency for Different Dimensions

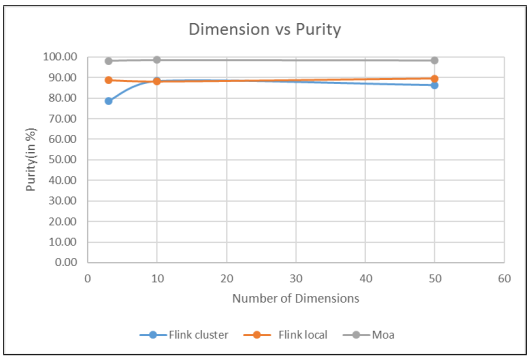

FIGURE 5.20: Purity for Different Dimensions

Next we compare the performance of our implementation against MOA for different number of clusters. As can be seen from figure 5.21, the CluStream implementation ran on the cluster performs well along with MOA.

At the end, we compare the latency result for different file sizes. The graph for this is a bit different as we tested our implementation on the cluster for higher file sizes, [100 MB, 512MB, 1024 MB], whereas on the local machine we restricted

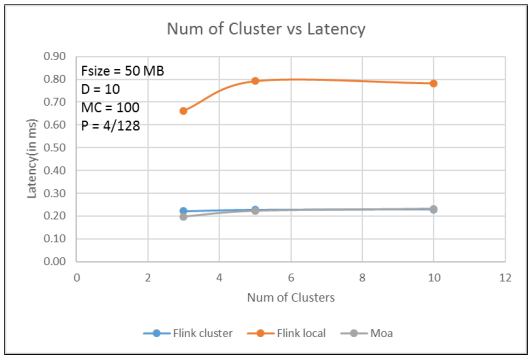

FIGURE 5.21: Latency for Different Number of Clusters

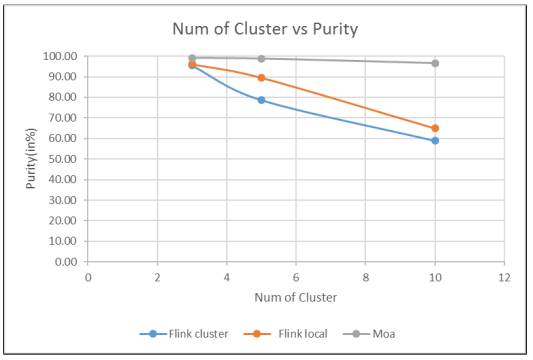

FIGURE 5.22: Purity for Different Number of Clusters

ourselves till 100 MB as the execution on the local machine for higher file sizes was not suitable.

As can be seen from figure 5.23, we can see that even with higher file sizes(which results in more data in the data stream), the latency keeps constant on the cluster whereas MOA gets a bit high on the latency.

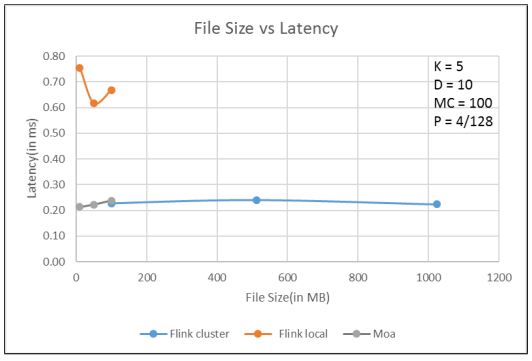

FIGURE 5.23: Latency for Different FileSize

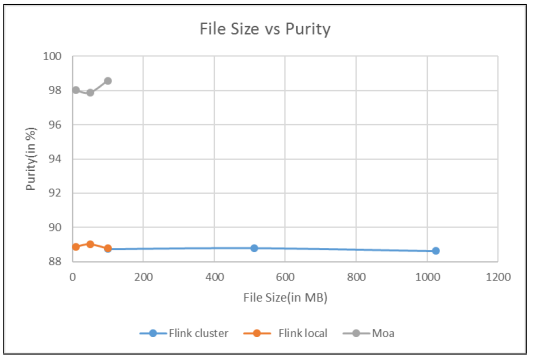

FIGURE 5.24: Purity for Different FileSize

The purity values for MOA in each of the comparison cases outperforms our implementation of CluStream, both on local as well as the cluster. In certain case like in figure 5.22, the purity goes down to 60%. This is a reason for concern and needs to be taken care of in the future.

Result Conclusion

To conclude, we have successfully shown that CluStream can be scalable and performs moderately well in the given test scenarios. There's obviously scope for

improvement, but the initial proof of concept is ready and data stream algorithms which follow concepts similar to CluStream can be scaled in a similar way.

Chapter 6

Conclusion

The thesis comprehensibly surveyed some of the existing state-of-the-art algorithms for clustering data streams and presents a comparative study of the same, so that the user can read and identify which algorithm can fit their problem and use the corresponding algorithm accordingly. All the algorithms surveyed claimed that they are scalable but the experimentations were done on a single machine with no parallelism which made this thesis more challenging as implementation of any of the surveyed algorithm required understanding not only of the algorithm but also of the underlying architecture of the parallel processing engine.

For parallel processing, we chose Flink which is a stream processing engine having API's in Java and Scala with python API in beta. Flink introduced additional challenges as some of the features needed to directly implement the algorithm chosen for implementation(CluStream) were not available and we had to employ workarounds to get the desired results.

We achieved the goal to survey data stream clustering algorithms and successfully implemented CluStream in a distributed and scalable environment. We worked around the challenges of implementing a stream clustering algorithm and challenges presented by the stream processing engine (Apache Flink) itself. The results obtained from the experiments were encouraging as we obtained clusters of good quality and with low latency across various set of parameters. There is obviously scope for improvement and optimizations can be performed on the system to improve the results of clustering without compromising on the latency.

This thesis is thus, a proof of concept for implementation of stream mining algorithm in a scalable and distributed environment, which is still not popular in the main stream despite having many useful use cases.

6.1 Discussion and Future Works

Future Works include improving the performance of the implemented algorithm by finding and performing optimizations to the current framework. This would include finding a better way to create macro clusters by selecting right set of microclusters as centroids, so that the cluster purity can be increased. It would also be really interesting to extend the online-offline framework of CluStream to other algorithms (like D-Stream etc.) which uses similar basic concepts with a few modifications. At the same time, improvement to the concept drift capture framework is needed, so that it can catch the gradual concept drifts as well, and not discard such data points as outliers.

It would be also interesting to notice the improvement in the performance with optimizations in the Flink's stream processing engine along with the availability of the missing features, such that our implementation would work without any workarounds.

Also, in the future we would like to perform a cumulative comparison against other scalable stream processing engines, which was unfortunately missing from this work.

Bibliography

[1] Amazon AWS - Streaming Data. https://aws.amazon.com/streaming-data/. [Online; accessed 17-July-2016].

[2] Amineh Amini, Teh Ying Wah, and Hadi Saboohi. On density-based data streams clustering algorithms: A survey. *Journal of Computer Science and Technology*, 29(1):116–141, 2014.

[3] Jonathan A Silva, Elaine R Faria, Rodrigo C Barros, Eduardo R Hruschka, André CPLF de Carvalho, and João Gama. Data stream clustering: A survey. *ACM Computing Surveys (CSUR)*, 46(1):13, 2013.

[4] Sudipto Guha, Adam Meyerson, Nina Mishra, Rajeev Motwani, and Liadan O'Callaghan. Clustering data streams: Theory and practice. *IEEE transactions on knowledge and data engineering*, 15(3):515–528, 2003.

[5] Vijay Arya, Naveen Garg, Rohit Khandekar, Adam Meyerson, Kamesh Munagala, and Vinayaka Pandit. Local search heuristics for k-median and facility location problems. *SIAM Journal on computing*, 33(3):544–562, 2004.

[6] Kamal Jain and Vijay V Vazirani. Primal-dual approximation algorithms for metric facility location and k-median problems. In *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pages 2–13. IEEE, 1999.

[7] 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*, pages 81–92. VLDB Endowment, 2003.

[8] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. Birch: A new data clustering algorithm and its applications. *Data Mining and Knowledge Discovery*, 1(2):141–182, 1997.

[9] Marcel R Ackermann, Marcus Märtens, Christoph Raupach, Kamil Swierkot, Christiane Lammersen, and Christian Sohler. Streamkm++: A clustering algorithm for data streams. *Journal of Experimental Algorithmics (JEA)*, 17: 2–4, 2012.

[10] David Arthur and Sergei Vassilvitskii. k-means++: The advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1027–1035. Society for Industrial and Applied Mathematics, 2007.

[11] Philipp Kranen, Ira Assent, Corinna Baldauf, and Thomas Seidl. The clustree: indexing micro-clusters for anytime stream mining. *Knowledge and information systems*, 29(2):249–272, 2011.

[12] Pedro Pereira Rodrigues, Joao Gama, and Joao Pedro Pedroso. Odac: Hierarchical clustering of time series data streams. In *SDM*, pages 499–503. SIAM, 2006.

[13] Pearson's Correlation Coefficient(PCC). https://en.wikipedia.org/wiki/Pearson_product-moment_correlation_coefficient. [Online; accessed 11-July-2016].

[14] Hoeffding Inequality. https://en.wikipedia.org/wiki/Hoeffding%27s_inequality. [Online; accessed 11-July-2016].

[15] Yixin Chen and Li Tu. Density-based clustering for real-time stream data. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 133–142. ACM, 2007.

[16] Li Tu and Yixin Chen. Stream data clustering based on grid density and attraction. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 3(3):12, 2009.

[17] Jiadong Ren, Binlei Cai, and Changzhen Hu. Clustering over data streams based on grid density and index tree. *Journal of Convergence Information Technology*, 6(1), 2011.

[18] Feng Cao, Martin Ester, Weining Qian, and Aoying Zhou. Density-based clustering over an evolving data stream with noise. In *SDM*, volume 6, pages 328–339. SIAM, 2006.

[19] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, volume 96, pages 226–231, 1996.

[20] Xuan Hong Dang, Vincent Lee, Wee Keong Ng, Arridhana Ciptadi, and Kok Leong Ong. An em-based algorithm for clustering data streams in sliding windows. In *International Conference on Database Systems for Advanced Applications*, pages 230–235. Springer, 2009.

[21] Apache Flink. https://flink.apache.org/features.html, . [Online; accessed 15-July-2016].

[22] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, et al. The stratosphere platform for big data analytics. *The VLDB Journal*, 23(6):939–964, 2014.

[23] Apache Flink, WikiPedia. https://en.wikipedia.org/wiki/Apache_Flink, . [Online; accessed 12-July-2016].

[24] Moa Clustream Code. http://www.cs.waikato.ac.nz/~abifet/MOA/API/_clustream_8java_source.html. [Online; accessed 12-July-2016].

[25] Flink Iteration. https://ci.apache.org/projects/flink/flink-docs-master/api/java/org/apache/flink/streaming/api/datastream/DataStream.html#iterate, . [Online; accessed 08-June-2016].

[26] Jason Altschuler. KMeansPlusPlus. https://github.com/JasonAltschuler/KMeansPlusPlus/blob/master/src/KMeans.java. [Online; accessed 08-July-2016].

[27] Flink Dataset API. https://ci.apache.org/projects/flink/flink-docs-release-1.0/apis/batch/index.html, . [Online; accessed 08-June-2016].

[28] Flink DataStream API. https://ci.apache.org/projects/flink/flink-docs-release-1.0/apis/streaming/index.html, . [Online; accessed 08-June-2016].

[29] MOA (Massive Online Analysis). http://moa.cms.waikato.ac.nz/. [Online; accessed 19-July-2016].

[30] Weka 3: Data Mining Software in Java. http://www.cs.waikato.ac.nz/ml/weka/. [Online; accessed 19-July-2016].