An experimental evaluation of sliding-window algorithms for k-means clustering

Mallick, Satyaki

Award date:
2021

Department of Mathematics and Computer Science

An Experimental Evaluation of Sliding-Window Algorithms for k-Means Clustering

Master Thesis

Satyaki Mallick

Supervisor:
Mark de Berg

TU Eindhoven, September 2021

Supervisor
Prof. Mark de Berg

Defense Committee
Prof. Mark de Berg
Prof. Wouter Meulemans
Prof. George Fletcher

An Experimental Evaluation of Sliding-Window Algorithms for k-Means Clustering

Satyaki Mallick, Masters in Computer Science, DSiE, '19-'21, TU/e
1410881 - s.mallick@student.tue.nl

Abstract

An experimental analysis of k-means clustering in the sliding window model in two dimensional space is performed. We perform this experiment on algorithms presented by Ackermann et al. [1] and Youn et al. [22] for k-means clustering on streaming data and sliding window streaming. We experiment on the different parameters of the algorithms and decide on values which are ideal for a well distributed gaussian dataset, to give a good trade-off between quality of clustering and space usage. We then go forward to also devise an algorithm which combines concepts from aforementioned algorithms and then analyze its parameters to find optimal values to use for clustering on a well distributed gaussian dataset.

1 Introduction

With the advancement of technology, data is now being generated at an enormous rate. Depending on the source, the entire dataset could be available before the start of computation. On the other hand, the entire dataset may not be available before computation starts. Such data is continuously generated at such a rate that all of it cannot be stored before processing. Moreover, the amount of data being generated is so big that it is impossible to store all of it. Thus, processing has to be done on the fly while only storing a subset of the data. This is called streaming data.

Business use cases involve calculating statistics like mean, frequent items, heavy-hitters, number of clusters etc. on the data. Calculating such metrics over the data give useful insights but the volume of the data gives new challenges. Algorithms on non-streaming data have access to all data throughout the entire computation, whereas streaming algorithms do not because they can only store a subset of the data.

Algorithms for streaming data generally can 'see' a datapoint only once. Moreover, unlike static data, where the data follows a pre-determined distribution, the distribution of data in streams may change over time, in particular when we also have sliding window, as discussed next.

Data streams treat all elements equally, i.e., there is no difference between an element arriving at the beginning of the stream and an element arriving at the end of the stream. But in practice, it is often the case that recent elements are much more important than the older ones. A way to model this is to consider the last w elements of the stream. Generally this w is much smaller than the total size of stream but is still big enough that all w elements cannot be stored in memory. Hence, the goal is to design algorithms with space sub-linear in w. Such kind of model was first introduced by Datar and Motwani in [12] and has been studied extensively in paper by Borassi et al[4].

1.1 Previous Work

In this thesis, we are interested in sliding-window algorithm of clustering. Intuitively, clustering a set of items is to partition the set into clusters such that items within the same cluster are "similar" to each other. Clustering can be broadly classified in the following 5 models:

- Hierarchical Clustering

- Centroid-based Clustering
- Distribution-based Clustering
- Density-based Clustering
- Grid-based Clustering

In this thesis, we discuss various centroid-based clustering problems. Before beginning, let us define some notations.

Given a set P of points in any metric space and a value $k < |P|$, the objective is to find k clusters $C_1, \ldots, C_k$. Each cluster C_i is defined by a cluster center c_i (or *centroid*, as it is often called). Given the set of centroids $c_1, \ldots, c_k$, the cluster C_i consist of all points in S for which c_i is the nearest centroid (typically with ties broken arbitrarily). The goal is now to chose the centroids such that some cost function is minimized. Below, we show the cost function for different centroid-based clustering methods. In the below expressions, $\text{dist}(\cdot, \cdot)$ denotes the distance between two points in the metric space.

- k-center: The cost is the maximum distance of any point from its cluster center. More precisely, the cost of the clustering defined by centroids $c_1, \ldots, c_k$ is defined as $\max_{1 \leq i \leq k} \max_{p \in C_i} \text{dist}(p, c_i)$.
- k-means: The cost is the sum of squared distances between points and its corresponding cluster centers. More precisely, the cost of the clustering defined by centroids $c_1, \ldots, c_k$ is defined as $\sum_{1 \leq i \leq k} \sum_{p \in C_i} \text{dist}^2(p, c_i)$.
- k-median: The cost function is the sum of distances between points and its corresponding cluster centers. More precisely, the cost of the clustering defined by centroids $c_1, \ldots, c_k$ is defined as $\sum_{1 \leq i \leq k} \sum_{p \in C_i} \text{dist}(p, c_i)$.
- k-medoid: The cost function is identical to k-means but the cluster centers are limited to one of the input points.

Before we move on with k-means let us briefly discuss some progress in k-center and k-median.

k-center clustering:

Hochbaum snd Shmoys [18] and Gonzalez [13] were the first to work on k-center clustering and achieved a 2-factor approximation in the cluster radius. The k-center problem has also been studied in the streaming model. Cohen-Addad et al. [10] gives a $(6 + \varepsilon)$ approximation for the k-center problem in generic metric spaces by only storing $O\left(k\varepsilon^{-1} \log \alpha\right)$ points in working memory where alpha is the ratio between the maximum and minimum distance between any two points in the point set and ε is a number between 0 and 1. The only drawback here is it assumes prior knowledge of α. Ceccarello et al. [8] gives comparable results to Cohen but uses significantly less memory and time for the case of k-center with Outliers by McCutchen and Khuller [21]. The Pelizzoni et al. paper uses ideas from the Cohen et al. [11] to create coresets for the k-center problem in doubling dimension and gives a $2 + \varepsilon$ solution using $O\left(k \log(\alpha)(c/\varepsilon)^D\right)$ working memory where $c > 1$ is a constant and D is the doubling dimension. Goranci et al. [14] gives $2 + \varepsilon$ approximation to the k-center problem given the underlying metric has a bounded doubling dimension and the aspect ratio is bounded by a constant. This algorithm is the current state of the art in terms of solution quality and running time though it does not give any space improvements over previously known algorithms.

k-median problem

Charikar et al. [9] proposed the first constant factor approximation to the problem for an arbitrary metric space using a natural linear programming relaxation of the problem followed by rounding the fractional solution. The first streaming solution for k-median problem was given by Datar, Motwani and Callaghan in [6] using $O\left(\frac{k}{\tau^4}W^{2\tau}\log^2 W\right)$ space with a $O\left(2^{O(1/\tau)}\right)$ approximation parameter where W is the window size and a user-defined parameter $\tau \in \left(0, \frac{1}{2}\right)$. The open question left unanswered here was can this be improved to poly-logarithmic space. Braverman et al. [6] gave the first polylogarithmic space approximation. In particular, they show that using only polylogarithmic space they could maintain a summary of the current window from which they can construct an O(1)- approximate clustering solution.

k-means clustering

The first k-means clustering algorithm was a heuristic algorithm developed by Lloyd et al [19]. [3] improved Lloyd's algorithm by changing the initialization technique to choose every subsequent point to be the farthest from the previously chosen one. This algorithm gave a bound on the error was also faster and more accurate compared to Llyod's algorithm. This algorithm is generally called k-means++ in the literature.

Streaming Model: Sequential k-means [20] is the first algorithm known which maintains current clusters and runs Lloyd's algorithm for each new point received. This was later complemented by BIRCH [23] to work on large datasets. Clustream [2] creates subsets of streams and run weighted k-means algorithm to find clusters. STREAMLS [15] gives a constant-factor approximation algorithm using a divide-and-conquer method which makes query time slow and hence is not suitable for fast query response. Har-Peled and Mazumdar in [17] has presented coresets of size $O\left(\varepsilon^{-d}k\log n\right)$ which was further improved to $O\left(kd/\varepsilon^6\right)$ by Har-Peled and Kushal in [16]. An algorithm by Ackermann et al. called StreamKM++ [1] improves then known running time for large number of cluster centers. This run-time was further improved by Zhang et al. in [24] using caching which is a novel approach in streaming clustering.

Sliding Window Streaming Model: In the real world, in a data stream, recent data is generally more relevant than old data. Hence there has been numerous attempts to implement streaming algorithm for k-means in a sliding window setting. Braverman and Ostrovsky [7] gives one of the first sliding window algorithm for k-means clustering provide a sampling schema which gives an optimal sampling schema that requires $O(k)$ space for fixed windows. In a different paper, Otrovsky and Braverman [5] also give another technique called smooth histograms which reduces the approximation error in exponential histograms. Later, Braverman et al. [6] improves on above algorithm presenting the first polylogarithmic space $O(1)$-approximation to the metric k-median and metric k-means problems in the sliding window model. Their algorithm uses $O\left(k^3\log^6 W\right)$ space where W is the window size. This paper was later improved by Borassi et al. [4] on generic metric spaces using space linear in k which is the current state-of-the-art. An independent line of research is continued by Youn in [22] which uses heuristics to reduce space and time and gives approximate guarantees but applies only to Euclidean spaces.

1.2 Our Contribution

k-means algorithm is the most widely used clustering algorithm in practice. In streaming setting and in sliding window setting, Ackermann's algorithm [1] and Youn' algorithm [22]

are the state-of-the-art respectively in terms of space usage at the time of doing this thesis. Thus, in this thesis, we study these two algorithms in detail. We also run experiments on Youn's algroithm to find parameter values of Youn's algorithm for good quality clustering on two-dimensional gaussian datasets. We find that a coreset size of 200 for a window size of 10000 is a good value for clustering. We also find that a higher threshold distance has a direct impact on clustering quality. Moreover, we also realized that a slide length of 1/10 of window size is gives stable cost curve. A value less than that deteriorates quality and a value higher than that makes the cost curve spiky. Then we create a sliding-window version of the Ackermann's algorithm which we call the Combo Algorithm using ideas from Youn's algorithm. Specifically, Combo Algorithm is made by running Coreset Tree of Ackermann's algorithm in every pane of Youn's algorithm. We test this algorithm for different values of slide length and find that an upper limit of coreset using four times more space, gives only about 6% improvement in quality compared to lower limit of coreset. Then we test both the algorithms on different distributions of data and find that for closely packed datasets, Youn' algorithm performs better although combo algorithm performs better otherwise.

2 Algorithms

In this section we describe the two algorithms that we will experiment with. Both algorithms try to construct a small coreset of the points in the stream. This is a subset of the points (possibly weighted) such that the centers of a good k-means clustering on the coreset is also a good set of centers for all the points. (Formally, a coreset should come with a guarantee on the quality of the solution computed on the coreset, but the algorithms below do not give such a guarantee.)

2.1 The algorithm of Youn et al. [22]

The main idea in this paper is the use of Group Feature (GF). We will explain these technique for the k-center problem with outliers in $\mathbb{R}^d$. Let P be a set of points in $\mathbb{R}^d$. Let $p := (x_1(p), \ldots, x_d(p))$ be a point in P. For a set $S \subset P$, we define Group Feature (GF) as a 4-tuple (LS(S), SS(S), N(S), T(S)), where each of the four elements is an aggregation of certain properties of P. More precisely, these elements are defined as follows:

- A Linear Sum term of points in S, denoted by, $\mathrm{LS}(S) = \left(\sum_{p\in S} x_1(p), \ldots, \sum_{p\in S} x_d(p)\right)$.
- A Squared Sum term of the points in S, denoted by, $\mathrm{SS}(S) = \left(\sum_{p\in S} x_1^2(p), \ldots, \sum_{p\in S} x_d^2(p)\right)$.
- The number of points in S, denoted by, $\mathrm{N}(S)$.
- The timestamp of the most recent of the containing points, $\mathrm{T}(S)$ where timestamp is the time at which a point arrives in the stream.

GFs for two sets can be merged the following way. Namely, for two sets $S_1, S_2 \subset P$, we have:

- $\mathrm{LS}(S_1 \cup S_2) = \mathrm{LS}(S_1) + \mathrm{LS}(S_2)$,
- $\mathrm{SS}(S_1 \cup S_2) = \mathrm{SS}(S_1) + \mathrm{SS}(S_2)$,
- $\mathrm{N}(S_1 \cup S_2) = \mathrm{N}(S_1) + \mathrm{N}(S_2)$,
- $\mathrm{T}(S_1 \cup S_2) = \max(T(S_1), T(S_2))$.

Each of the above is a constant time operation. Thus, merging of two GFs takes constant time. For a set Q containing a single point $q := (x_1(q), \ldots, x_d(q))$, its properties can be defined as follows:

- $\mathrm{LS}(Q) = (x_1(q), \ldots, x_d(q))$

- $\mathrm{SS}(Q) = \left(x_1^2(q), \ldots, x_d^2(q)\right)$
- $\mathrm{N}(Q) = 1$
- $\mathrm{T}(Q) = \mathrm{T}(q)$

Thus updating a GF to merge a single point to it is a constant time operation. Additionally, the value of the centroid of a GF can be calculated from the GF properties as $\mathrm{LS}(S)/\mathrm{N}(S)$. The idea will be to use a collection of GFs as a coreset (although strictly speaking it is not, as a GF is not just a weighted point).

The streaming setting applied in this paper is different from the general sliding window setting. That is, instead of one point entering the window at a time, this paper considers slide length, where L points enter the sliding window at once. Under such a setting, the whole window of length W is divided into $Z := \lceil W/L \rceil$ *panes*. A single such pane can contain a maximum of L points. The goal of the algorithm is to create a coreset of size (roughly) M for the points inside the window where M is a parameter that can be set by the user. The elements of the coreset will be GFs. Thus in each pane, $m := M/Z$ GFs need to be generated.

Algorithm 1 ConstructCoreset

```
1: procedure ConstructCoreset(stream P, coreset size m, θ)
2:     S ← empty coreset
3:     θ ← determine the maximum distance between two points to include in same GF
4:     for each b ∈ P do
5:         GF_s ← nearest GF in S to b
6:         if dist(b, GF_s) < θ then
7:             update GF_s with new item b
8:         else
9:             create new GF_b based on b.
10:            S ← S ∪ {GF_b}
11:        if |S| >= 2m then
12:            S ← ReduceCoreset(S, m)
```

At any point, as points are generated, GFs for the most recent pane are continuously being created and updated by Algorithm 1. If a new point is not sufficiently close to an existing GF, it gets its own GF, otherwise, the point is absorbed in an existing GF. When $2m$ GFs are created, Algorithm 1 calls Algorithm 2 which merges $2m$ GFs back to m GFs using previously mentioned merging techniques. Thus, at any point, there could be m to $2m$ GFs in memory.

The above process continues and Algorithm 1 keeps seeing new points, creating GFs from them and adding them to the global coreset. After L points are seen, the algorithm drops the oldest pane and its containing GFs from the global coreset (global coreset refers to the coreset of the whole window).

Algorithm 2 ReduceCoreset

```
1: procedure ReduceCoreset(Coreset S, coreset size m)
2:     Q ← empty set
3:     R ← S
4:     for each GF_s ∈ S do
5:         R ← R - {GF_s}
6:         if not first iteration then
7:             GF_q ← nearest GF in Q to GF_s
8:         GF_r ← nearest GF in R to GF_s
9:         if dist(GF_s, GF_q) < dist(GF_s, GF_r) then
10:            delete existing GF_q from Q
11:            new GF_q ← Merge GF_q and GF_s
12:            insert new GF_q in Q
13:        else
14:            delete existing GF_r from R
15:            new GF_r ← Merge GF_r and GF_s
16:            insert new GF_r in Q
17:        if |Q| + |R| ≤ m then
18:            Q ← Q ∪ R
19:            break
20:    return Q
```

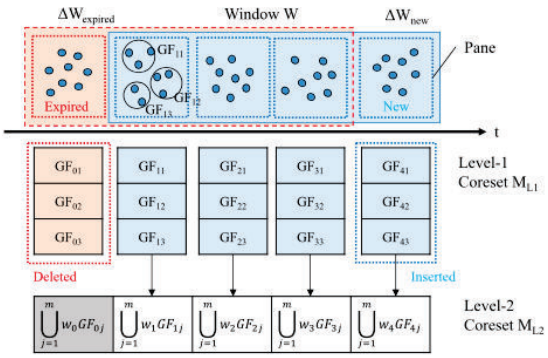


Figure 1 Figure showing coreset formation in a pane and in the whole window. Taken from [22]

These sets of GFs in each pane constitute the level-1 coresets. A level-2 coreset is generated by the union of all the GFs in Level-1 while also multiplying each one of the GFs with a corresponding weight factor. This is demonstrated in Figure 1. The goal of the weight function is to give different priorities to different points in the window. One way to do this is: older panes get assigned a smaller weight than the more recent ones. Or, in a different approach, all windows could be assigned a constant weight. For our experiments, we have this weight set to 1.

Given, there are $O(m)$ GFs in each pane and Z panes, their union gives a coreset of size $O(M)$ again. When queried, a k-means clustering algorithm is run on the coreset to give k cluster centers.

Speeding up the algorithm by Locality Sensitive Hashing

As can be seen from Algorithm 1, when a new points arrives, it finds the GF nearest to it. For each of the n points in the stream, the algorithm searches for the nearest GF among m GFs, and since the points are d dimensional, this search operation takes $O(dmn)$ time for all points in total. As can be seen, this operation is quite expensive. Algorithm 2, the algorithm takes $O(dm^2)$ which is also expensive.

To counter this, the authors came up with Locality Sensitive Hashing (LSH) technique. In this, the points which are closer than a certain threshold map to the same hash value with a large probability.

A hash function $h_{\vec{a},b}(\vec{x}) : \mathbb{R}^d \to \mathbb{N}$ is a scalar projection that maps a vector $\vec{x}$ to an integer. A global hash function is defined as a concatenation of values from different hash functions. For example, if we have hash functions $h_1(x), h_2(x) \dots h_z(x)$, a global hash function will be $(h_1(x), h_2(x) \dots h_z(x))$. Using a global hash function increases the probability of points which are close by to be mapped to the same hash value.

When a new point arrives, a decision is made whether this point should be absorbed by one of the existing GFs or a new GF should be created. This decision is made by making use of hash values. To elaborate, the hash value of the point is created by passing the point through the global hash function. If the hash value already exists in the table, the corresponding GF is updated with this new point. If the hash value does not exist in the table, a GF is created by assuming the point as a singleton set and generating the GF properties (as described above). Then the (hash value, GF) pair is stored in the hash table. As we notice, the hash value of a GF is determined by the first point creating the GF though ideally, the hash value should be calculated from the center of the GF. But the authors accept this error margin for ease of implementation.

This technique is not applied in our implementation as we are not optimizing for running time in this thesis. Further details about implementation and running time improvement with this technique can be found in [22].

2.2 StreamKM++: A Clustering Algorithm for Data Streams [1]

The main idea of this paper is the use of CF trees and merge-and-reduce technique in form of buckets. Both of these concepts will be discussed but before we understand them, we need to understand what sampling with probability D^2 means in the context of StreamKM++.

For any two points $x, y \in \mathbb{R}^d$ and any set of points $C \subset \mathbb{R}^d$, we denote the Euclidean distance between x and y by $\mathrm{D}(x, y) = \|x - y\|_2$, and we define

$$\mathrm{D}(x, C) = \min_{c \in C} \mathrm{D}(x, c)$$

Similarly, for squared Euclidean distances, we define

$$\mathrm{D}^2(x, y) = \|x - y\|_2^2 \quad \text{and} \quad \mathrm{D}^2(x, C) = \min_{c \in C} \mathrm{D}^2(x, c).$$

The phrase, *chosen randomly with probability* D^2 is used in the context of k-means++ seeding procedure. In this procedure, during initialization a set of points among the input points needs to be chosen. In this set, the first point is chosen uniformly at random. The next subsequent points are chosen randomly with a probability proportional to the sum of squared distance from the closest point already chosen, i.e., *chosen randomly with probability* D^2.

The coreset construction strategy

From a point set P, such that $|P| = n$, we choose a set of points $S = \{q_1, q_2, ...q_m\}$. The first point q_i is chosen uniformly at random from P and inserted into S. For next subsequent point $p_i \in P$, define $D_i := \frac{D^2(p_i,S)}{\sum_{p_i \in P} D^2(p_i,S)}$. Now select every point p_i with probability D_i and put the selected points in a set S, thus choosing points farther from the previously chosen points with a higher probability than ones which are not. Let Q_i be the set of points in P closest to q_i. To each such q_i, we assign a weight equal to $|Q_i|$.

After choosing a sample point q_i, while choosing the next sample point q_{i+1}, we need to measure the distance of every point in P to it's nearest center in S. This takes $\Theta(dnm)$ time. But this can be sped up by a data structure introduced in this paper called the coreset tree. This data structure is described below.

The Coreset Tree

In a Coreset Tree, the point set P is split into subsets and assigned to the nodes of a tree. Instead of calculating pair-wise distances among the whole point set, now, distance measurements are limited to the subset of points present in a particular node. Since we attempt to maintain a balanced binary tree (by method described below), any kind of distance measurement takes only $\Theta(d \log m)$ time.

A coreset tree T is a binary tree that is made by dividing the point set P at each level. We start with a single cluster of the whole set P at the root. As we go down each level in the tree, we divide the cluster associated with the root into two sub-clusters such that they are "far away" from each other. More precisely, a coreset tree for a point set P is defined as follows.

- Every node v in the tree has an associated point set P_v, which is a subset of P; a representative point q_v from P_v; a cost equal to the sum of squared distances from q_v to each point in P_v for a leaf node and the sum of costs of its children for an internal node.
- The root of the tree T is associated with a cluster consisting of the whole point set P.
- Every leaf l consists of a subset of the point set P_l, and and a representative point q_l for the subset. Every representative point q_l has an associated weight which is equal to $|P_l|$.
- The two child nodes of a node v correspond to the two subsets generated from the point set P_v associated with v.
- The point set associated with a node are only stored for the leaf nodes.

Algorithm 3 describes the method of inserting a point into the coreset.

Algorithm 3 Insert a new point from the Stream

```
1: procedure InsertPoint(point p, bucket size m)
2:     put p into B_{-1}
3:     if B_{-1} is full then
4:         create empty bucket S
5:         move points from B_{-1} to S
6:         empty B_{-1}
7:         i ← 0
8:     while B_i is not empty do
9:         U ← B_i ∪ S
10:        S ← Reduce the size of Coreset U to m by creating m leaf nodes of the Coreset Tree as
   explained below.
11:        empty B_i
12:        i ← i + 1
13:    move points from S to B_i
```

Coreset Tree Construction. To ensure that the points chosen for the set S are far away from each other, the following strategy is chosen. The first representative point q_i for $i = 1$, is chosen at random. For the next point, q_{i+1} onwards, the following happens:

The Coreset Tree pseudocode is as follows:

1. For a node u, which has 2 child nodes, for each of its child node v_j for $j \in \{1, 2\}$, the probability of choosing v_j is proportional to $cost(v_j)/\sum_{j=1,2} cost(v_j)$.
   Now, this $cost(v_j)$ is calculated as follows: For node v_j, let us assume it's point set as P_v and representative point as q_v. Now,

$$\text{cost}\,(v_j) = \sum_{p_v \in P_v} \text{weight}\,(p_v)\,\text{dist}\,(q_v, p_v)$$

2. A child node of u is chosen randomly with this probability repeating the process until a leaf node is reached. Let this leaf has the associated point set P_l and the representative point q_l.
3. Now in P_l, we choose a new sample point q_{i+1} randomly with probability proportional to its distance from q_l.
4. Now we divide the point set P_l into two subsets and assign each of its point to q_l or q_{i+1} depending on which is closer. From the two created subsets, each one is assigned to child nodes of l, namely l_1 and l_2.
5. We update the cost of node l as the sum of costs of node l_1 and l_2.

The Merge and Reduce technique

In order to use the coreset tree technique described above in streaming settings, one need to account for the infinite stream of data. This is done by the merge-and-reduce technique.

The merge and reduce works under the following assumptions, although, the coreset from the coreset tree is not guranteed to have the these properties.

- If S_1 and S_2 are (k, ε) -coresets for disjoint sets P_1 and P_2, respectively, then $S_1 \cup S_2$ is a (k, ε) -coreset for $P_1 \cup P_2$.
- If S_1 is a (k, ε) -coreset for S_2 and S_2 is a (k, ε') -coreset for S_3, then S_1 is a $(k, (1+\varepsilon)\,(1+\varepsilon') - 1\,)$-coreset for S_3.

The merge-and-reduce technique maintains $L = \lceil \log_2(n/m) + 2 \rceil$ buckets. These buckets can be generated on the fly, so, there is no requirement to know the values of n or m

beforehand. Each of these buckets can either be filled completely or absolutely empty. They cannot be half-full. Any bucket B_i represents $2^i m$ input points (except the first bucket B_{-1}). Each of these buckets will have a capacity of m points which we will call the *bucket size*.

Connecting Merge-and-Reduce technique with Coreset Tree. Let us explain this with an example:
Let us imagine all the buckets are empty and first m points of the stream arrive and are stored in bucket B_{-1}. As the next m points of the stream arrive, we notice that bucket B_{-1} is full. Thus, contents of bucket B_{-1} is moved to bucket B_0 and the new m points are stored in B_{-1}. Now, the next set of m points arrive, but both bucket B_{-1} and B_0 are full. Thus points in B_{-1} and B_0 are merged using the coreset tree. To do so, a union of the points in B_{-1} and B_0 are treated as the root of the tree. Then a coreset tree is constructed on this root branching at each node until m leaf nodes are created. The representative points from these m nodes form the coreset representing $2m$ input points. This is denoted by the weight of these m points which always sum up to $2m$ for bucket B_1. These points are then copied to B1 emptying B_{-1} and B_0. This process is repeated for every higher level bucket.

Thus, any bucket B_i represents $2^i m$ input points and the sum of weights of its m points always sum upto $2^i m$.

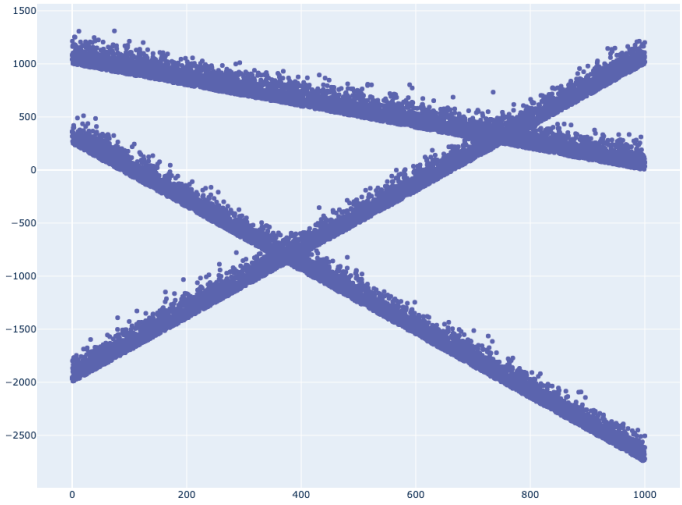
3 Datasets and Experimental Setup

3.1 Synthetic Data

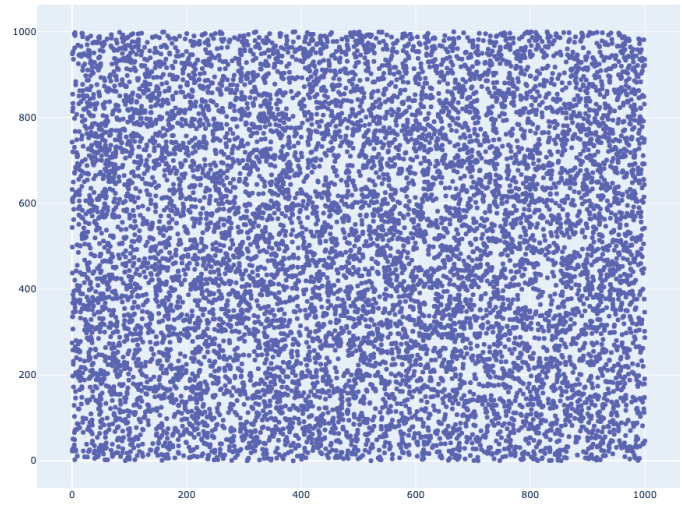
For creating synthetic dataset, scikit-learn's inbuilt function *make_blob()* was used. The version of scikit-learn used was 0.24.1. The points arrive uniformly at random in the stream. The following types of synthetic datasets were generated.

Note: Box: In the text below, multiple mention of the word *box* will be noticed. It is the distance between which all the points of the dataset can be found.

- **S1:** 2D points in three straight lines with a gaussian variance around it (15,000 points inside a box of 1000 x 5000 and 5000 points in each line). There was no specific way of choosing the lines. We chose three lines randomly with slope and intercept: (3, -2000), (-1, 1000), (-3, 250) and generated points along it. A standard deviation of 100 was used on one side of each line.

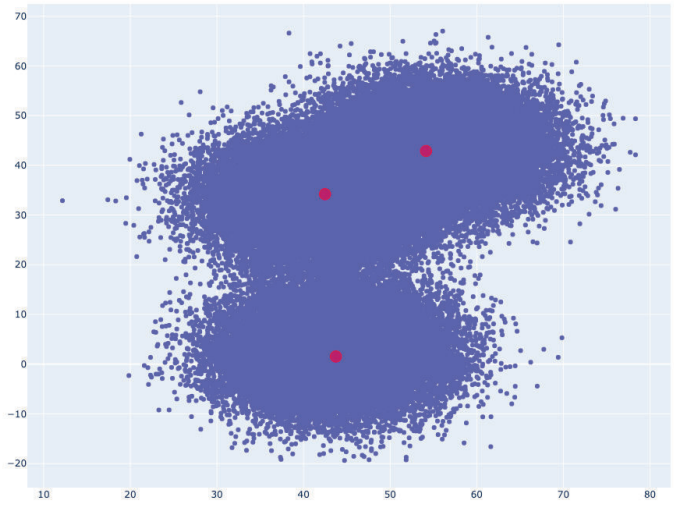


- **S2:** 2D points uniformly distributed in the space (150,000 points inside a box of 1000x1000).



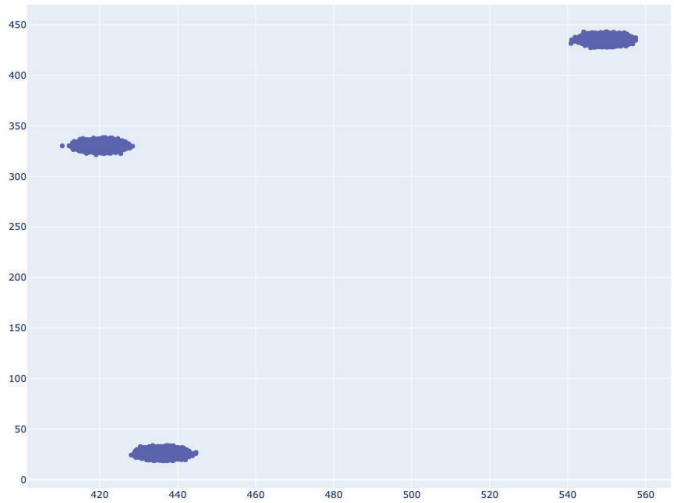
- **S3(q):** 150,000 2D points with q gaussian clusters closely lying beside each other (standard deviation of 6, and box of 100 x 100). The centers are chosen at random based on a *random_state* parameter of make_blob() function which was set to 2.

  Below is an example for $q = 3$.



- **S4(q):** 150,000 2D points with q gaussian clusters well-separated from each other (standard deviation of 2, and box of 1000 x 1000). The centers are chosen at random based on a *random_state* parameter of make_blob() function which was set to 2.

  Below is an example for $q = 3$.



- **S5:** 15,000 2D points with 3 gaussian clusters well-separated from each other (standard deviation of 6, and box of 100 x 100). This dataset is only used for experiments in section 4.2.1 where the focus is on the behaviour inside a window and not on the volume of data.

3.2 Implementation

All the experiments were performed on a 2017 Macbook Air with 1.8 GHz Dual-Core Intel Core i5 processor, 8 GB 1600 MHz DDR3 RAM, Intel HD 6000 Series 1.5 GB Graphics and 128 GB Solid State Drive.

Moreover, all the code was run in python 3.8.8.

For implementation of k-means and k-means++, scikitlearn's inbuilt function, *KMeans()* was used.

For many of the experimental results, the y-axis of the plots represents *relative cost*. This is the ratio of the algorithm cost to the optimal cost. The algorithm cost is computed by, first, finding the coreset over the input stream, running k-means++ on the coreset, getting k centers, then calculating the clustering cost over the original input using these k-centers. The optimal cost is calculated by running k-means++ directly on the input stream.

4 Experimental investigation of the GF-based algorithm

In this section we experimentally investigate the algorithm by Youn et al. [22]. From now on, we will refer this algorithm as the GF-based algorithm.

In order to visualize what GF-based algorithm is doing under the hood, we experiment with a S3(3) dataset in streaming setting with windowing.

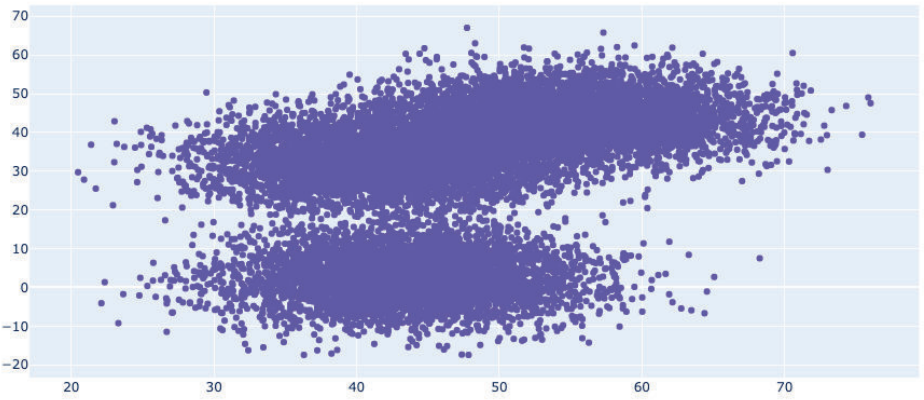


Figure 2 Original Input

Figure 2 shows the original input points.

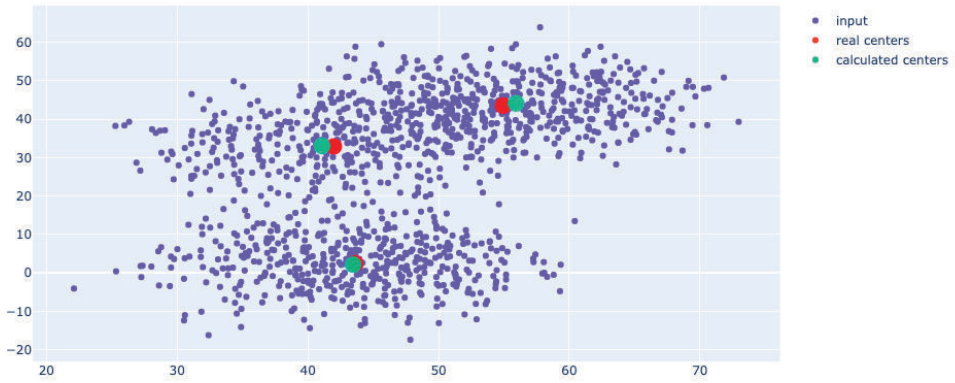


Figure 3 Original Centers and Generated centers using algorithm

Figure 3 shows the coresets generated by the GF-based algorithm for one of the windows using $W = 5000$, $L = 1000$ and $M = 300$ and the centers found on the coreset using the k-means++ algorithm (in cyan) and original centers (in red). As can be seen, the calculated centers and the generated centers are close by.

This was a short primer on how the GF-based algorithm or any coreset-generating algorithm works in general. They try to generate a subset of points from the input which is a good representation of the original dataset. In the next section, we will go deeper and experiment the effect of different parameters in the GF-based algorithm.

In this algorithm, we have the following variables that can be optimized:

- Slide Length L. Recall that the GF-based algorithm partitions the window into panes. The number of points in each pane is called the *slide length* and it is denoted by L.
- Coreset Size M. It is a parameter that tells the total number of coreset points that the algorithm must generate in a window.
- Threshold distance θ. The parameter that specifies the distance less than which a point is absorbed in an existing GF.

Window number : For many of the plots the x-axis corresponds to window number, i.e, the current window being processed if the windows are numbered from 1 at the beginning of the stream. This is determined in the following way: For a window W, number of points n, and slide length L, there are total of n/L panes in the whole stream and every time a window slides by one pane. Thus the total number of windows in the stream will be n/L. But the window can only slide up to $n/L - x$ panes where $x = W/L$, because the window must contain W points. So, window number $= n/L - x$. Taking as an example, if a stream consist of $n = 150,000$ points, and $L = 40$, $W = 1000$, then $x = 25$. Thus, there are a total of $15000/40 - 25 = 350$ windows in the stream, and window number runs from 1 to 350.

4.1 Analyzing Cost in-between Slides L of the Window

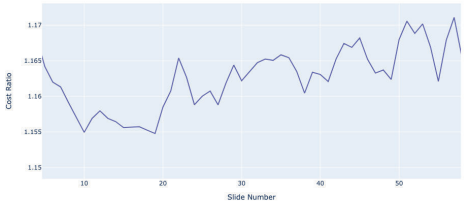
In this experiment, we are going to investigate what happens when the oldest pane of a window is dropped and a new pane enters the window. During this, L new points enter the window one at a time and are processed into GFs.

Our hypothesis is as soon as a pane is dropped, all the corresponding GFs drop and the quality of clustering falls. Then as new points enter, new GFs are created and the clustering

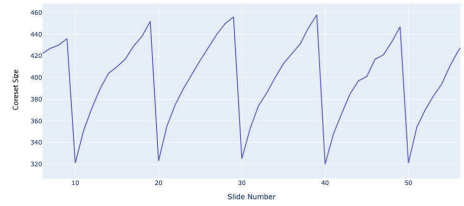
quality improves. This improvement continues till L points are seen. Immediately after, the quality drops again as the oldest pane in the window is dropped along with all the GFs inside it. So, after every L points we expect to see an incease in clustering cost and a drop in coreset size. In order to test this, we take a reading of the cost and the coreset size after every $L/10$ points are seen. This $L/10$ is termed as in-between-slide value now on.

To perform this experiment, the window size is set to 1000, in-between slide is set to 1/10th of L. We are using the smaller S5 dataset because we are interested in what happens inside a pane of a window. So, a large dataset isn't necessary and will make the plot cluttered. $M = 200$ was used (as of why will be explained in a later section) and a θ of 2 is used (as of why will be explained in a later section).

We will perform the experiment for 2 slide lengths:
(i)$L_1 = W/2$,
(ii)$L_2 = W/10$

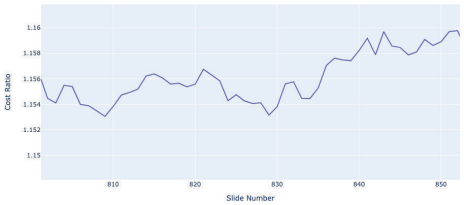


(a) Cost ratio vs. Slide number

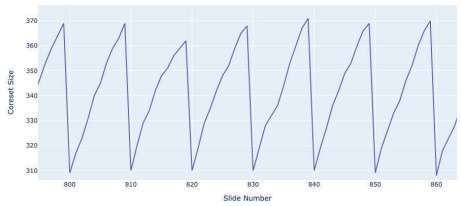


(b) Coreset Size vs. Slide number

Figure 4 Variation for L = 1/2 W. (The y-axis indicate the actual coreset size used by the algorithm and not the parameter M)



(a) Cost ratio vs. Slide number



(b) Coreset Size vs. Slide number

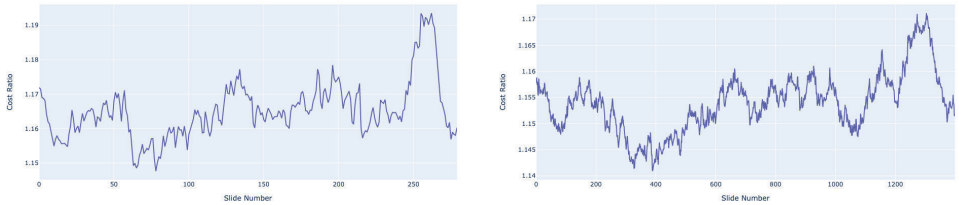
Figure 5 Variation for L = 1/10 W, (The y-axis indicate the actual coreset size used by the algorithm and not the parameter M)

Figure 4 and Figure 5 shows a portion of the experiment ran on the whole stream. The y-axis denote the relative cost. The x-axis denote the slide number, i.e., how many in-between slide intervals have passed since the start of the stream. Figure 4 shows the experiment for $L{=}W/2 = 500$. Therefore, in-between-slide is 50. Thus, every 10th reading on the x-axis indicate that one pane is complete. Every 11th reading indicates start of a new pane. Figure 5 shows the experiment for $L{=}W/10{=}100$. Here also, every 10th reading on the x-axis indicate that one pane is complete, and 11th pane indicate start of a new pane.

A few observations can be made based on the plots above:

- **Global shape of cost curve over time for L_1 compared to L_2:** Ideally the cost curve for both L_1 and L_2 should be very similar (shown in Fig 6) with comparable peaks and troughs. This is because the number of coreset points is pre-defined for a window. Thus, after the end of each slide, there is always same number of coreset points both for L_1 and L_2 with a variation of M to $2M$. Apart from that, only cost variations should be seen during the pane generation.

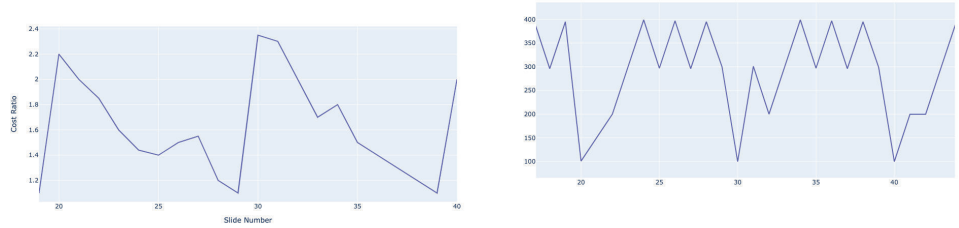
  We indeed notice very similar cost curves although the one for L_2 is 0.02 less than the one for L_1.



(a) L1 **(b)** L2

Figure 6 Comparing Cost Curve for $L1$ and $L2$

- **Bigger spike in $L1$ compared to $L2$:** The cost curve for L_1 should show bigger spikes compared to the cost curve for L_2. This is because, after each slide, a lot more coreset points are lost, thus losing information required for clustering. When L is $1/2$ of W, $1/5$th more information is lost compared to when L is $1/10$th of W. If we look at Figure 4a and Figure 5a, we notice such a behaviour.
- **Cost curve vs the coreset size curve:** Figure 4 and Figure 5 shows a comparison of cost variation with changing coreset size. We notice that the x-axis value at which the cost is the minimum, the coreset size is also the largest. In the immediate next value of x-axis, the coreset size drops and thus the clustering cost increase.
- **Behaviour when $L = W$:** Now, we are going to perform a new experiment where we will set $L = W$. The algorithm slides one full window at a time and we expect to lose all the coreset points when this happens thus much bigger cost spikes. As we see in Figure 7a, $L = W$ causes bigger spike, with the cost reaching 2 times OPT at the peak of the spike. Since $L = W = 1000$ for this experiment, in-between slide is 100. Figure 7b shows that the coreset size drops to around 100 after every slide. This means approximately every point forms a GF of its own. And then this coreset size keeps increase upto around 400 after which the coreset is reduced again. This goes on at the end of each slide.



(a) in-between slides cost variation for $L = W$.

(b) in-between slides coreset size variation for $L = W$. (The y-axis indicate the actual coreset size used by the algorithm)

Figure 7 Comparing Cost and Coreset Size Curve for $L = W$

Thus, we notice that a smaller value of L results in more stable cost curve. We choose $L = 1/10$ of W for our future experiments because that will give stable cost curve in a practical use case. Although, for our experiments, the value of L will not impact the cost curve because we only calculate cost readings at the end of a slide. We have not tested for

values of L less than $1/10W$ although experiments in Section 6 shows the impact of using such smaller values.

4.2 Influence of Coreset Size M on Overall Cost of Clustering

In this experiment we are going to find a good coreset size for clustering in well-separated gaussian datasets by the GF-based algorithm. We are going to run experiments on two different clusters with two different values of k. We are going to use S4(3), S4(6) datasets. We are going to set $W = 10000$ and $L = 1000$. A θ of 2 was used for this experiment.

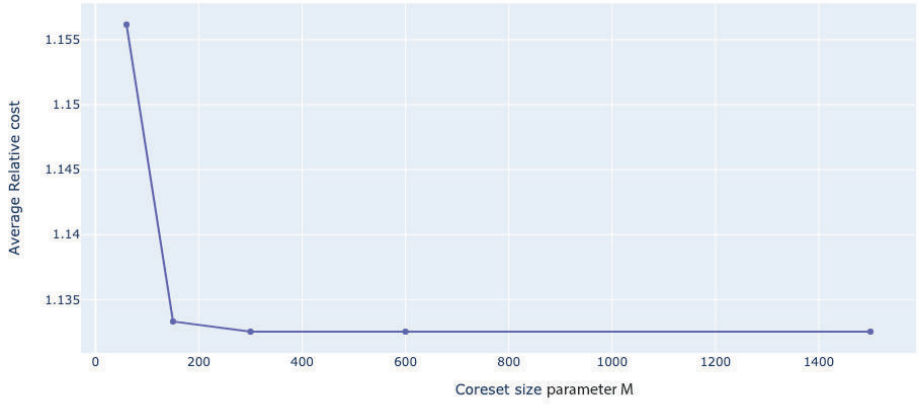


Figure 8 Average relative cost vs defined coreset size for S4(3) for $k = 3$

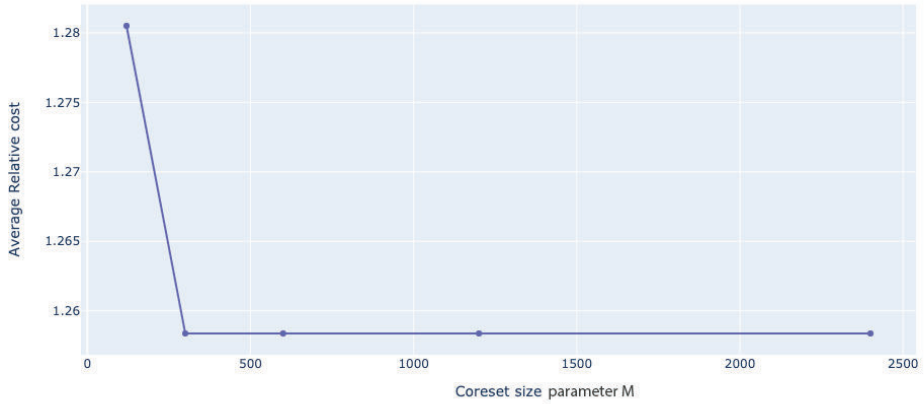


Figure 9 Average relative cost vs defined coreset size for S4(6) for $k = 6$

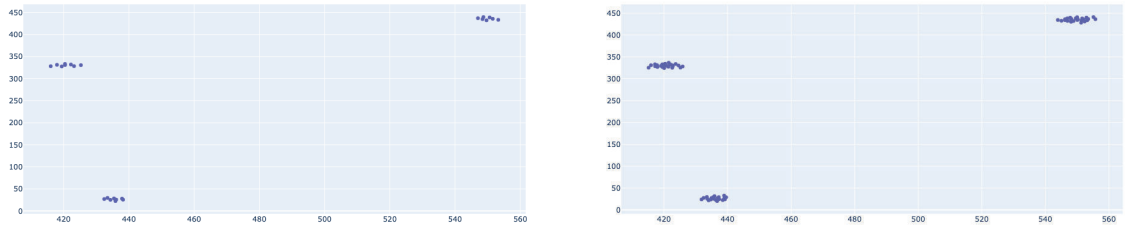
The y-axis represent the average relative cost and the x-axis represents the parameter M set in the algorithm. As we notice from Figures 8 and 9, the cost stabilizes around 200 to

300 coreset points. Although, we must notice that the difference in cost is very small. For example, in Figure 9 the cost ratio is roughly 1.28 for the first measurement and roughly 1.25 for the second measurement. For future experiments, we will keep our coreset size between 200 and 300. This is also close to the value of M found by Ackermann and GF-based paper (100 times the value of k).

Analyze the distribution of Coreset for different values of M and Quality Impact of the Chosen Value of k

The goal of a coreset is that it approximates the distribution of the points in the stream well, so that solving the k-center problem on the coreset gives a solution that is also good for the whole stream. However, it could be that a coreset is good for a certain value of k and not so good for another value of k. In particular, it could be that the coreset is not so good when the value k that is used in the clustering does not correspond to the number of clusters in the data. The goal of the next experiment is to investigate this.

In the next figures, we are going to visualize the coreset generated by the minimum value of M used (60) and the maximum value of M used (1500) in the experiment of Figure 8.



(a) M=60 **(b)** M=1500

Figure 10 Generated coreset for one window on S4(3) dataset

Looking at figure 10, the coreset generated for $M = 60$, and $M = 1500$, we see that the coreset points are almost equally divided among all clusters. Precisely, cluster 1 has 24, cluster 2 has 35 and cluster 3 has 26 coreset points. For Figure 10b, cluster 1 has 671, cluster 2 has 543 and cluster 3 has 686 coreset points. Figure 10 is for one single window but we can expect a similar behaviour in all windows because the points in the input stream are randomly distributed. S4(6) dataset also shows similar results. From this visualization, we can conclude that the value of M on this dataset do not affect the distribution of the generated coreset compared to the original input distribution and thus the value of chosen k will not have a significant impact on clustering quality, i.e., if the number of clusters in the dataset is previously unknown, then choosing a higher value of k would always reduce the cost of clustering and not the other way round.

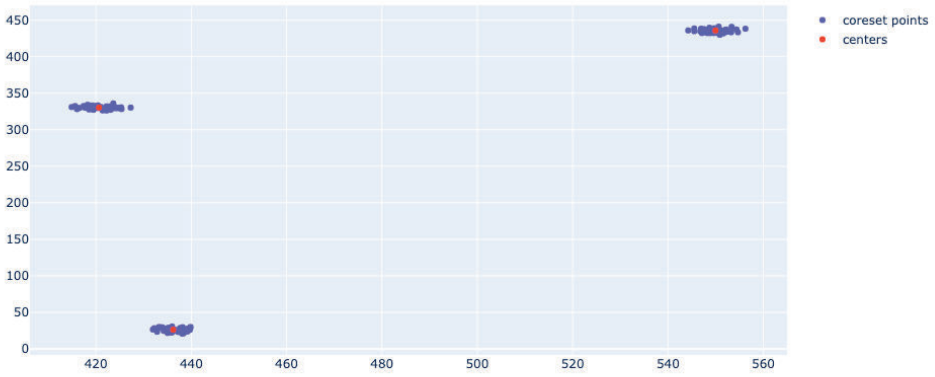


Figure 11 Generated coreset for one window on S4(3) dataset. $k = 3$

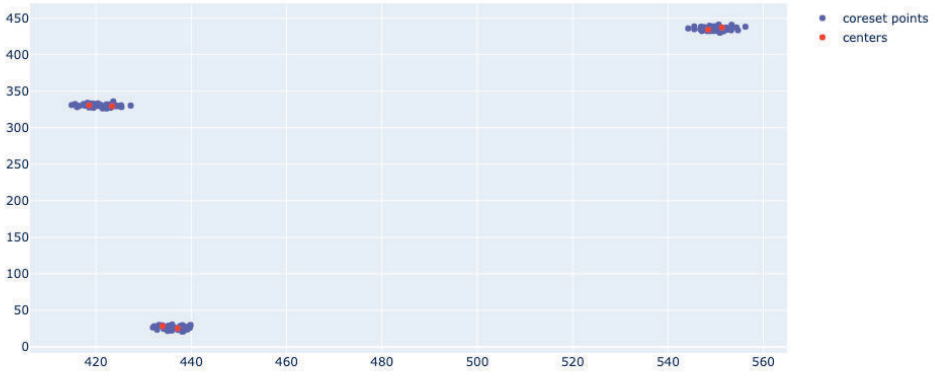
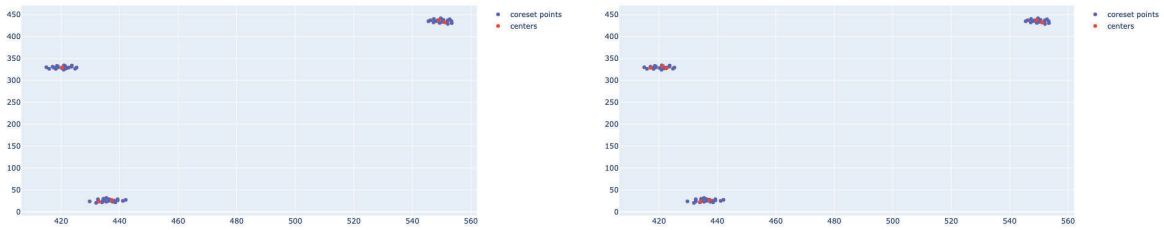


Figure 12 Generated coreset for one window on S4(3) dataset. $k = 6$



(a) k=5 **(b)** k=7

Figure 13 Generated coreset for one window on S4(3) dataset

Running k-means++ on the coreset of Figure 10b for $k = 3, 5, 6, 7$ generates the centers shown in Figure 11, 12 and Figure 13. Figure 14 shows relative cost with progressing time. There is a definite drop in the average cost. This is because the excess centers (2 excess centers for $k = 5$ and 4 excess centers for $k = 7$) are generated inside the clusters in order to minimize cost.

As we notice, the generated coreset in Figure 10 is a very good representation of the underlying distribution of the dataset for well separated clusters. Thus, a higher value of k can only make clustering quality better and not worse.

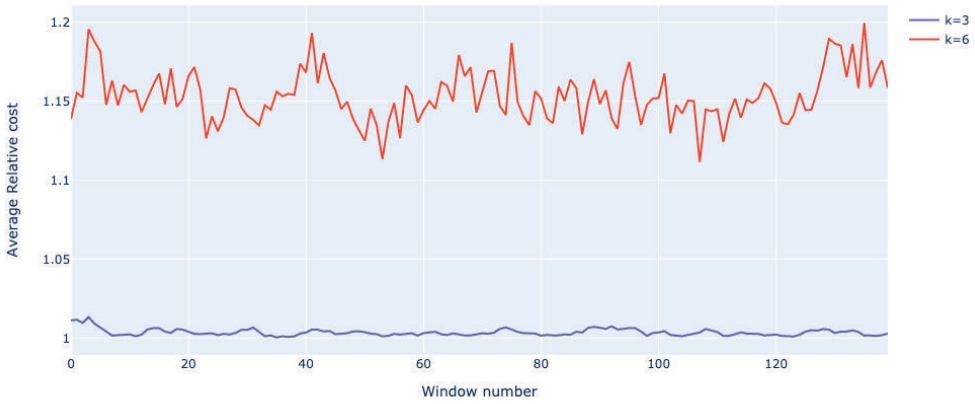
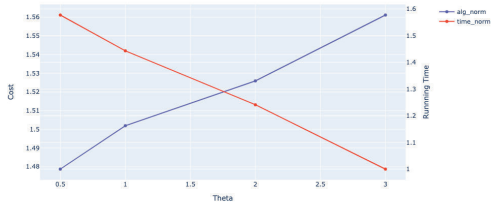


Figure 14 Relative cost with progressing time in the stream for $k = 3$ and $k = 6$ on S4(3)

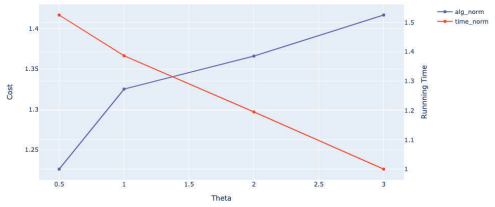
4.3 Influence of a Low Threshold value θ on Clustering Quality and Running Time

Now we will investigate the threshold distance θ on a the quality of clustering. θ is the maximum distance for which a point would be absorbed into an existing GF. For this, we will run the experiment on two types of datasets, one with S3(3) clusters and the other with S3(6) clusters. For each of the above two types of datasets, we will use three different boxes – 100, 1000, 10000. For the all datasets W was set at 10,000, L was set to 1,000 and M fixed

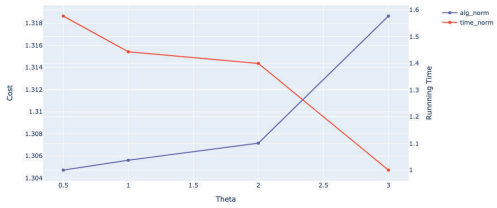
at 200. To note, that these experiments were run without varying k. So, for the datasets used, k in the algorithm was exactly set to the number of clusters in the dataset.



(a) Cost vs θ vs time for box of 100

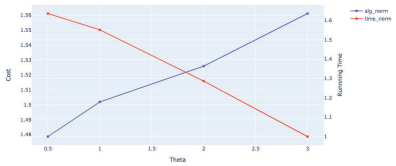


(b) Cost vs θ vs time for box of 1000

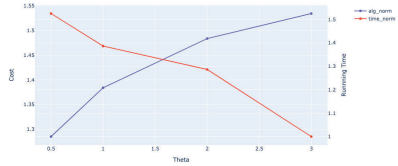


(c) Cost vs θ vs time for box of 10000

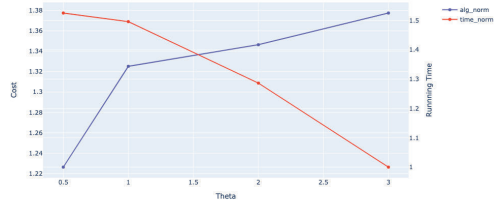
Figure 15 Figures for datasets with 3 clusters



(a) Cost vs θ vs time for box of 100



(b) Cost vs θ vs time for box of 1000



(c) Cost vs θ vs time for box 10000

Figure 16 Figures for datasets with 6 clusters

Every figure shows one x-axis that represents the value of θ and two y-axes, one representing the average cost of running the algorithm on all window on the left axis, the other represents the time taken to run the algorithm on the right axis. The blue line represents the cost of clustering using the algorithm. The red represents the running time of the algorithm.

The left y-axis is normalized with optimal cost and the right y-axis is normalized with the minimum among all running times.

There are a few points to remember here:

1. The coreset size M is pre-defined before running the algorithm, thus this parameter is fixed.
2. Each pane is configured to have m coreset points summing upto M coresets for the whole window.
3. Let us also note that a bigger coreset will give a better quality clustering but also result in a higher running time.

Therefore, if M is fixed, why do we see a variation in cost and running time with increasing θ (in reference to Figure 15 and Figure 16)? To understand this, we have to go back to how the GF-based algorithm works. If we notice, even though we specify a fixed M, and thus a fixed m, the algorithm allows upto $2m$ coreset points from each pane. Adding up coresets from all panes, this actually results in upto $2M$ coreset points in each window. Hence, the coreset size can vary between M and $2M$.

Improving quality with a smaller θ as seen in the figures above is a result of this variation in M. A smaller θ causes creating a coreset size tending towards $2M$ and a bigger coreset means a lower cost, thus, a better quality of clustering although running time increases because the k-means++ algorithm has to run on a larger set of points. A bigger θ gives a coreset size tending towards M, thus giving lower quality but better running time. The reason a smaller θ gives a coreset size tending to $2M$, is because a smaller θ causes most points to have a GF of its own thus creating a lot more GFs. Thus, the coreset of each pane will have a higher tendency to reach $2m$ points resulting in a larger overall coreset size.

As we notice both in figure 15 and 16, as θ increases, we see an increase in cost of clustering. This is also because a greater θ causes a greater approximation at the GF creation level, more points being absorbed in a particular GF. Let us investigate Fig 16c. We notice that when we increase θ from 2 to 3, there is a little increase in cost, but the running time decreases significantly. This plots give an idea of what kind of trade-offs we are ready to accept based on the problem at hand.

If θ is made even smaller (<0.5) upto zero, each point makes its own GF. But because M is fixed and pre-defined, reduce_coreset() function is called everytime $2m$ points are seen. At this point, the number of points seen and the number of GFs is the same. Thus, the running time increases significantly. But the clustering quality becomes high because the algorithm is even more likely to generate the upper limit of coreset size of $2M$.

Thus a conclusion can be drawn, that we should choose a value of θ depending on how much running time we can afford. In general, higher the θ, lower the running time. So, the approach should be to think about how much running time is acceptable and then pick the smallest θ that we can have.

5 Combo Algorithm

5.1 Strategy

Recall that the algorithm of Ackermann et al. [1] works in the insertion-only streaming model. The goal of this section is to extend the algorithm to the sliding-window setting, and to experiment with the new algorithm. Adapting the algorithm of Ackermann et al. will be done using the ideas from the algorithm of Youn et al. Therefore we will call the new

algorithm the *combo algorithm*. For this algorithm, the plan is to use the idea of windows and panes from GF-based algorithm but use Coreset Tree instead of their GF creation method to generate coreset in each pane. From an incoming stream of data, the algorithm processes only the points present in the current window. A window is divided into panes and the GF-based algorithm creates coreset on one pane at a time using GF creation method as explained before. It has a predefined coreset size M for the whole window, which is divided by the number of panes to get the number of coreset points that need to be generated for each pane, denoted by m. For a pane under construction, the algorithm keeps receiving points and creates GFs from them. After seeing L points, the algorithm saves the coreset generated so far and starts creating new GFs for the next pane. Once all the panes for a certain window have been created, k-means++ is run on the resulting coreset.

In our combo algorithm, we are going to replace the GF creation method with Ackermann's Coreset Tree technique to generate the coreset. To elaborate, for a given pane, the algorithm will keep receiving points and create coreset points from them using Coreset Tree. After seeing L points, the algorithm saves the coreset generated so far and starts creating a new coreset for the next pane. The number of coreset points that will be generated in a pane will be a factor of m, the bucket size defined in the Coreset Tree. The method of determination of the coreset size will be explained in a later section.

Whenever a user demands cluster centers, the algorithm merges the coresets of all the panes in the current window and runs k-means++ algorithm on the weighted coreset points. This returns k-centers. To evaluate the quality of clustering, we calculate the cost of these coreset calculated k-centers over the actual points in the window. To find the optimal cost, we run the k-means++ algorithm directly on the points in the window, same as what we do for GF-based algorithm.

To give a brief description of the Coreset Tree technique, it keeps taking input points and incrementally keeps filling up buckets of size m. At any point when a bucket is full, they are merged and moved to the next bucket increasing their weight. The merging is done using merge-and-reduce technique.

The most efficient use of space by the combo algorithm will be when all the coreset points are found in the highest level bucket created so far in a given pane and all the lower level buckets are empty. Since each bucket represents $2^i m$ input points, such a scenario will arise when L/m is an exact power of 2. The least efficient space usage will be when all buckets are full. This case will arise when $L = 2^i m - 1$ for some value of i. We test the algorithm for two extreme limits of coreset size: (1), Only the highest bucket is full and all other buckets are empty. We refer to this as the *lower limit of coreset size*. (2), All the buckets are full. We refer to this as the *upper limit of coreset size*.

5.2 Experiments

In this algorithm, we have the following variables that can be optimized:

- Slide Length L
- Coreset Size M of the Coreset Tree

5.2.1 Analysis of Cost Variation for Upper and Lower limit of Coreset Size over Time for a Fixed Bucket Size m While Varying L

For this experiment, we are going to analyze the different values of the coreset size in a pane. The dataset used was S4(3). Moreover, we defined $W = 10,000$ and k was set to 3.

In figure 17, the green curve represents the cost fluctuation for the lower limit of coreset size. It represents the cost when $L = 40$ and bucket size, $m = 5$. Note that, $2^i \cdot 5 = 40 \implies i = 3$. Hence, in the case of lower coreset size, we will have 4 buckets, B_0, B_1, B_2, B_3 in each pane with only the last bucket being full. This results in a total of 5 coreset points in a pane. Moreover, there are a total of $10000/40 = 250$ panes in a whole window. This makes total of $250 \cdot 5 = 1250$ coreset points in the window.

For a pane, the number of buckets cannot go higher than this, because as soon as L points are received, the buckets are reset for consuming points from the next pane.

Let us now investigate the fluctuation of cost for the upper limit of coreset size keeping the number of buckets same. For this we use $W = 10000$, $L = 79$, and we set m (the bucket size parameter) to 5. This ensures four buckets and all the buckets are full.

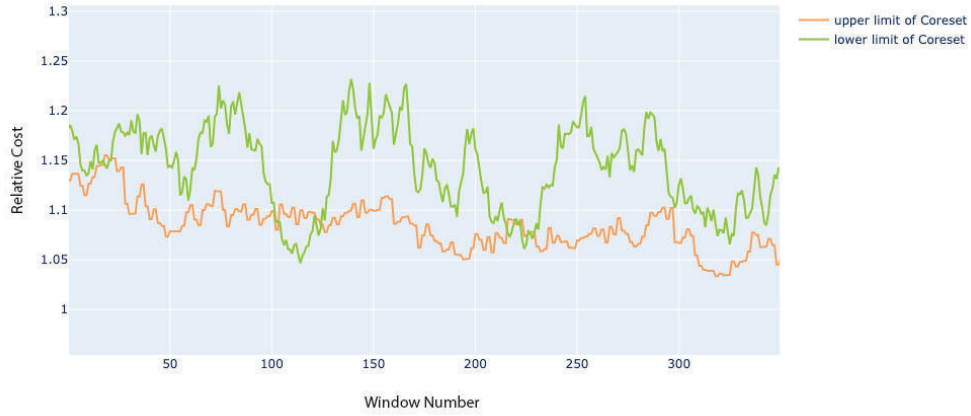


Figure 17 Fluctuation of cost for each window using upper limit of Coreset Size comparing with lower limit of Coreset size

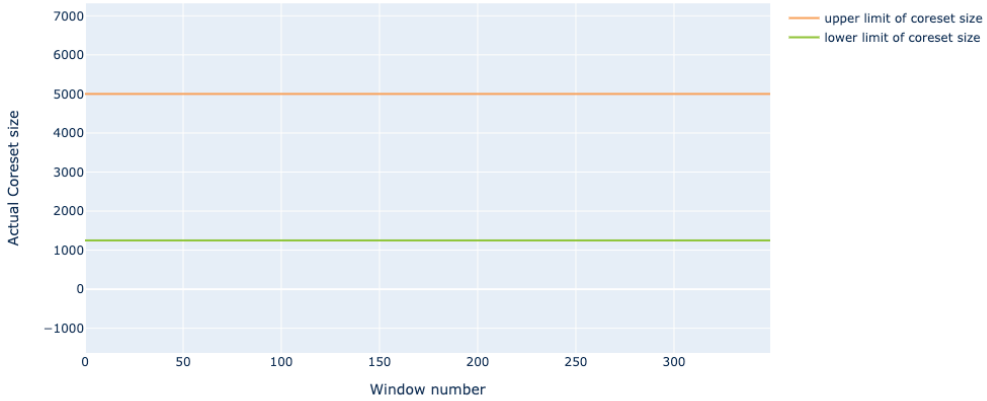


Figure 18 Coreset Size variations for upper and lower limit

In Figure 17, the orange curve represents the cost fluctuations for the upper limit of coreset size. We use four buckets for each pane but all the buckets are full. We notice that the average cost is lower. We also notice that with lower limit of coreset size, the cost is fluctuating much more. By using 4 buckets which are all filled, the algorithm uses four times the space representing 79 input points using 20 coreset points. (The other experiment was using 5 coreset points to represent 40 input points). Figure 18 shows the variations of coreset size for the upper and lower limits.

To measure how much quality improvement this setting gives, we find the average cost of running combo algorithm and the average optimal cost for both values of L and find the percentage increase in cost with respect to optimal cost. We find that the algorithm when using 4 times more space gives only 5% improvement in clustering cost.

To confirm the behaviour further, we used a bigger dataset, of 300,000 points with 6 well separated clusters and $k = 6$. This experiment shows that there is only a 6% increase in quality while using four times more space. Figure 19 shows that.

Figure 19 Relative Cost variation for dataset of size 300,000 (showing the first 250 window numbers)

5.2.2 Analysis of Average Cost of Clustering on the Whole Stream for a Fixed Bucket Size m While Varying L

Whenever we increase L/m by a power of 2, we increase the number of buckets in a pane by 1. In this experiment. We are going to investigate the effect of adding a bucket on the quality of the coreset. To measure quality improvement over the stream, we take the average of cost over all windows. For this experiment, we are going to use S4(3) dataset and $k = 3$. Let us also fix window size to 10000. Let us also fix bucket size at 5, and choose L as various multiples of 40. So, 40, 80, 160, 320, 640 etc. Each of these values causes the highest bucket number to increase by 1 and filling that bucket only whereas all lower level buckets are empty. Thus, there are always only 5 coreset points per pane. But each pane approximates more points.

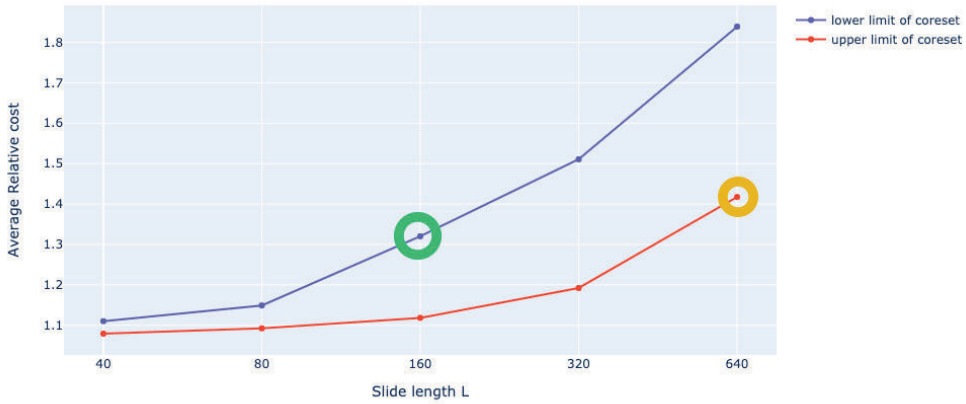


Figure 20 Comparing Average Cost among all windows for upper and lower limit of coreset size

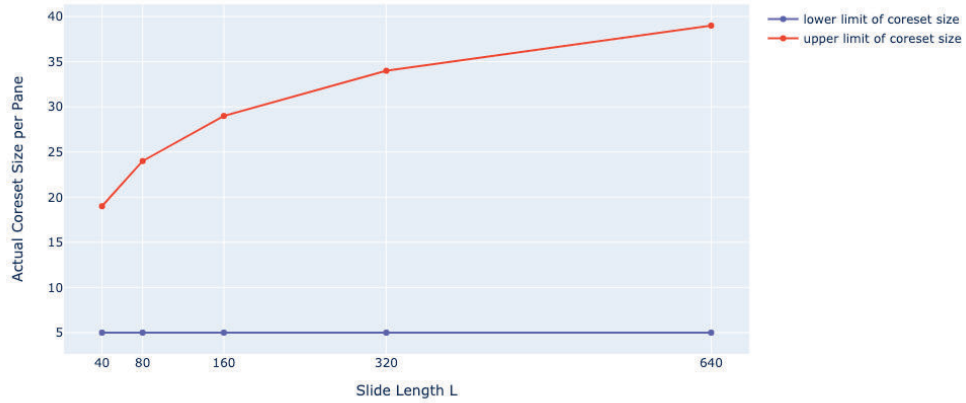


Figure 21 Actual Coreset size per pane vs Slide Length L

In Figure 20, the x-axis represents the value of L used and y-axis shows the relative cost. In the blue curve, we notice that the cost consistently increases with increasing L because higher the value of L, higher the number of input points that gets approximated by the coreset which remains fixed at 5. Thus, the same 5 points have to represent more input points causing a greater degree of approximation. Thus, a lower value of L gives better quality clustering on a well-distributed Gaussian dataset.

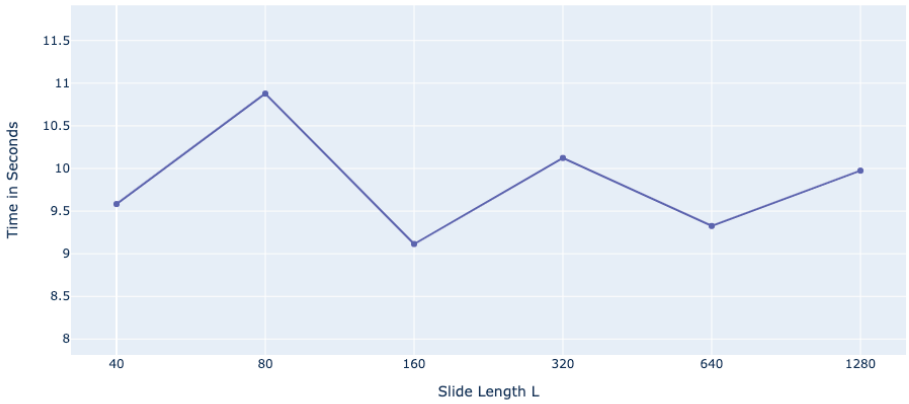


Figure 22 Total Running Time vs no of input points represented in the Coreset

But, this does not effect the coreset creation time because the bucket sizes are the same (only a few arraylist manipulations happen). This is shown by Figure 22 where x-axis represents the parameter L and the y-axis represents the average time taken by the combo algorithm to create a coreset of a window.

Now, we are going to test the algorithm for the maximum coreset size, that is all the buckets are full. For this test, we are going to use L as 1 less than the previous tested numbers, that is: 39, 79, 159, 319, 639. At these numbers, for bucket size 5, all buckets are full except bucket B_{-1}, where there are only 4 points. As we see in Figure 20 for the red curve, the upper limit of coreset plot is less steep compared to lower limit of coreset. This is expected because there are more coreset points available for the experiment, and the k-means++ algorithm has more data to work on.

Figure 21 shows the actual coreset size as slide length is increased (and resultant increase in number of buckets). It is interesting to see that at $L = 160$, lower limit of coreset size (green circle) provides a better quality than upper limit of coreset size at $L = 640$ (orange circle). But green circle actually uses only 5 coreset points whereas orange circle uses 39 coreset points. This could be because of the fact that at green circle, $L = 160$, thus, only 160 of the input points are seen by the algorithm and are summarized to 5 points. But, at the orange circle, $L = 640$, 640 of the input points are seen by the algorithm which is 480 points more summarized by 39 points only. This could be a possible reason for the difference in quality.

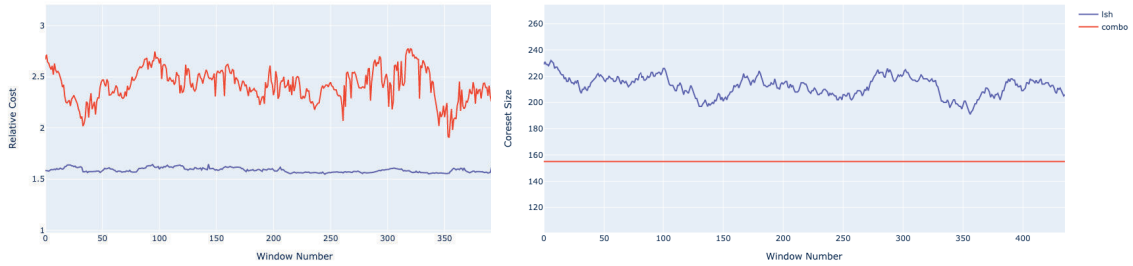
6 Comparing GF-based Algorithm and Combo Algorithm

In this section, we will compare the quality of clustering by the two algorithms, when the number of coreset points used by the two of them are comparable (though we say comparable, combo algorithm will always be using somewhat smaller coreset). Recall that m denotes the bucket size parameter of the combo algorithm. We will choose slide length L in such a way that every pane generates m coreset points. To keep it comparable with GF-based algorithm, the number of coreset points GF-based algorithm will generate in a pane is also set at m. This makes GF-based algorithm generate m to $2m$ points per pane. Multiplying

m with the number of panes, combo algorithm will generate M coreset points per window whereas GF-based algorithm will generate M to $2M$ coreset points per window. But, we experimentally observe, GF-based algorithm generates from M to a maximum of $1.5M$ coreset points per window. Unless otherwise mentioned, the value of k is set equal to the number of clusters in the dataset.

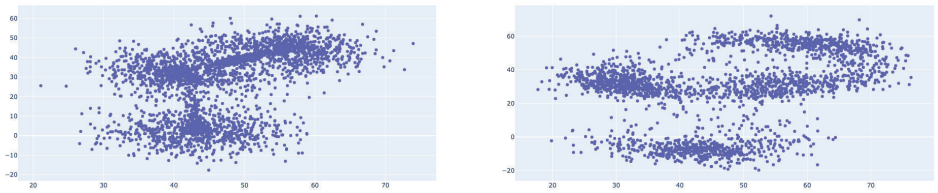
6.1 Comparing on S3 Dataset

In this section, we experiment on the S3(3) dataset with $W = 10000$ and $L = 320$. Therefore, there are a total of $10000/320 \approx 31$ panes. We set the bucket size parameter to 5. We use the lower limit of bucket size for this experiment thus each pane will have 5 coreset points for combo algorithm and thus $5 \cdot 31 = 155$ coreset points per window. To maintain a similar number of coreset points in GF-based algorithm, we set M (number of coreset points in a window) parameter of GF-based algorithm to 155. Thus, during runtime. GF-based Algorithm will have 155 to 310 coreset points per window. θ was set to 2.



(a) Cost vs. Window number **(b)** Coreset Size vs. Window number

Figure 23 Cost and Coreset size variation



(a) Coreset GF Algorithm **(b)** Coreset Combo Algorithm

Figure 24 Generated coreset by the 2 algorithms

Figure 24 shows generated coreset by the two algorithms. We notice, GF-based algorithm has a tendency to join the centers of the two algorithms (from Figure 24a). This happens when the value of L is small compared to the window size. Here, for $W = 10000$ and a well distributed dataset, points from all clusters arrive in the window of 10,000 points. But after seeing every 320 points, the algorithm has to reduce them to 5 points. This may cause merging of points from two different clusters to create a coreset point. The way GF-based

algorithm merges points is by taking the linear sum of the points and then dividing it by its weights. This causes the coreset points to appear on the straight line joining the centers of two clusters.

Moreover, we also notice that combo algorithm is finding more coreset points on the outskirts of the cluster (from Figure 24b). This is probably because of the coreset tree technique. At every node in the tree, the algorithm tries to find a point furthest from the currently chosen points. Suppose the tree receives 10 points from 3 different clusters, it will try to choose 5 coreset points among them which are the farthest from each other. These points may be farthest when they are on the outskirts of the distribution. This causes the coreset points to concentrate at the periphery of the input distribution. Because the coreset points are not a good representation of the input distribution, as we see in Figure 25 and the calculated centers are much skewed compared to actual centers.

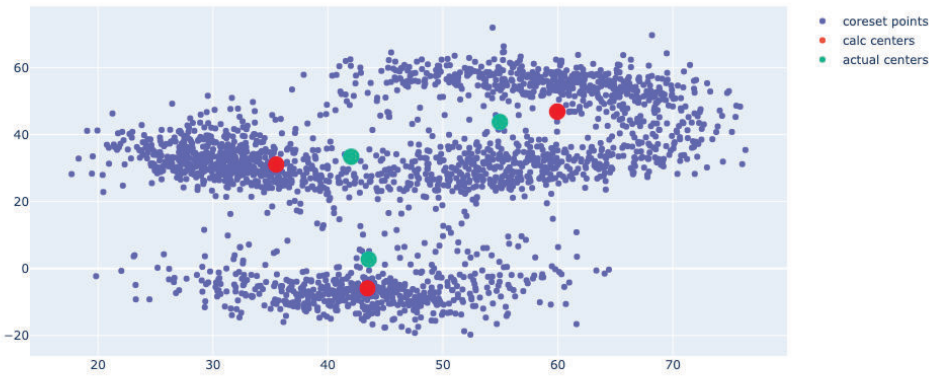


Figure 25 Combo Algorithm calc. centers vs actual centers

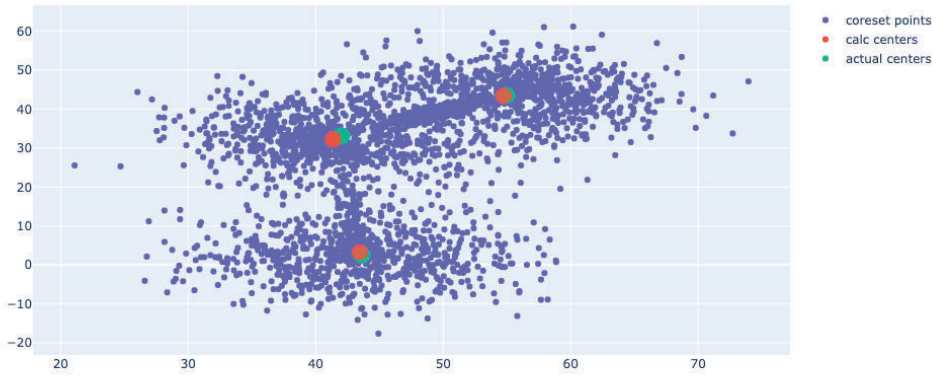


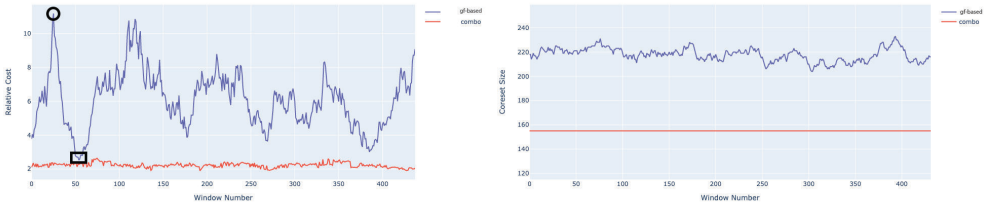
Figure 26 GF-based Algorithm calc. centers vs actual centers

Comparing this with GF-based algorithm, we see in Figure 26, the calculated centers are quite close to the actual centers. As a result, we notice, Combo Algorithm has a much higher average cost compared to GF-based algorithm, shown in Figure 23a.

6.2 Comparing on S4 Dataset

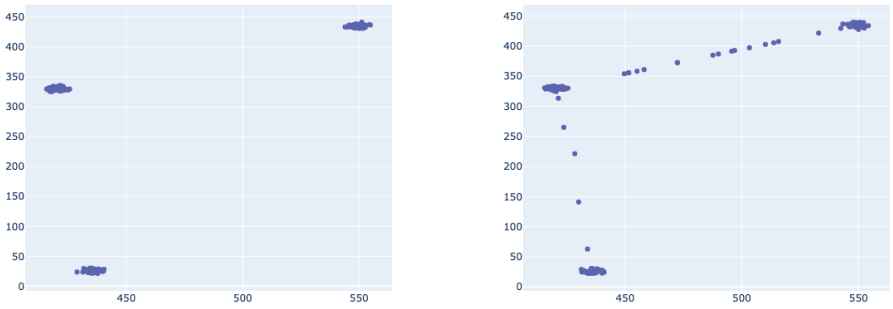
We repeat the above experiment on S4(3) dataset with $W = 10,000$, 150,000 points and $L = 320$. θ is set to 2. Figure 32a shows the relative cost with progressing window and Figure 32b shows the variation in coreset size with the progressing window.

We find an interesting observation in Figure 32a. Unlike the previous experiment, the average cost of the GF-based algorithm is higher than that of combo algorithm. The Reduce_Coreset() greatly deteriorates the quality of clustering as it generates points where there are no clusters by merging points from two different clusters by means of averaging. Recall that Reduce_Coreset() is the subroutine in GF-based Algorithm which gets called when the number of GFs generated in a pane exceeds the predefined value of coreset size in a pane(m) by 2 times.



(a) Cost vs. Window number **(b)** Coreset Size vs. Window number

Figure 27 Cost and Coreset Size variation with Window number



(a) Points received after one slide

(b) Coreset points generated after one slide

Figure 28 Actual Input and Coreset points generated by GF-based Algorithm for a window

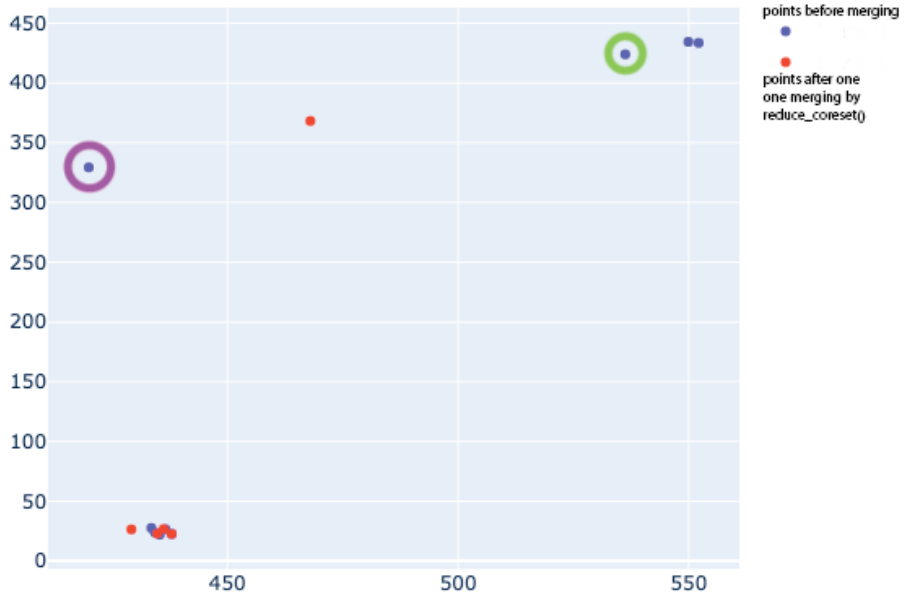


Figure 29 GF-based algorithm generate coreset points outside input clusters

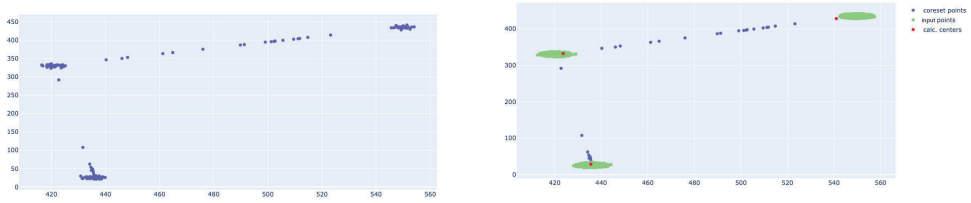
The observation in Figure 28 shows that the algorithm generated coreset points on the hypothetical straight line joining two clusters. To further illustrate, in Figure 29, blue dots

represents the GFs after 42 points are seen, and 9 coreset points are generated. Red dots represents the Gfs after 43 points are seen, 10 coreset points generated and reduced to 5 coreset points. Here, the highlighted purple GF shouldn't be merged with any GF because that is the only representative of the cluster there. But, since the algorithm forces merging, it takes the GF closest and merges with it, thus moving the average somewhere in between them. In this case, the green highlighted GF is the closest, thus, the purple coreset point merges with it and forms the center red GF lying in between the green and the purple ones.

The value of L has an influence on coreset-generating behaviour. A large value of L causes lot more coreset points to be generated. But, since the value of m is fixed, the GFs need to be reduced often by calling Reduce_Coreset(). Since, this sub-routine is the main quality-reducing factor, more calls to it can lead to a bad coreset.

Another possible explanation of such a behaviour is too small a coreset size, which forces points from different clusters to merge together. This can be fixed by increasing the coreset size or by reducing the value of L such that less GFs are created and less GFs need merging.

To note that such observation was not seen in Section 4.2.2. This is because the slide length was larger($L = 1000$) whereas here $L = 320$. The choice of $L = 320$ was made to keep the coreset size comparable between the two algorithms.

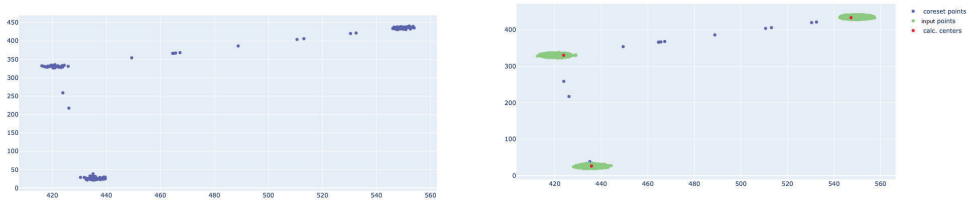


(a) Coreset points in a window

(b) Coreset points, centers and actual input in a window

Figure 30 Analysis of one of the maxima of GF-based Algorithm cost curve

If we look at Figure 32a at the circle marker, the cost is the maximum for GF-based algorithm. Looking at the coreset formed, and the centers generated in Figure 30a and Figure 30b, we see the the right-most generated center is outside the cluster. This causes high increase in cost.



(a) Coreset points in a window

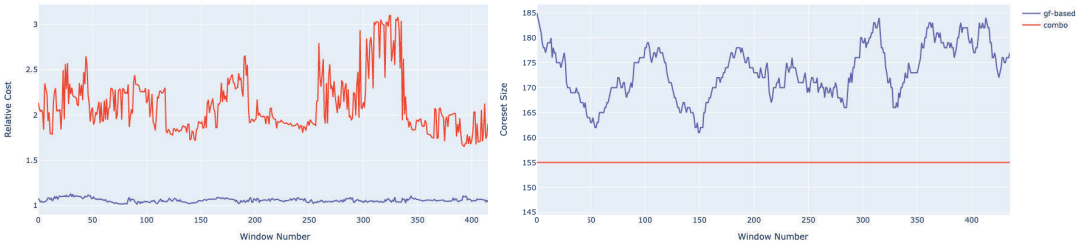
(b) Coreset points, centers and actual input in a window

Figure 31 Analysis of one of the minima of GF-based Algorithm cost curve

If we look at Figure 32a at the square marker, the cost is the minimum for GF-based algorithm. Looking at the coreset formed, and the centers generated in Figure 31a and Figure 31b, we see that all centers are approximately at the center of the clusters. Hence, the cost drops.

6.3 Comparing on S2 Dataset

For this experiment, a window size of 10,000 was used. θ was set to 2. L was set to 320 and the coreset size per pane varies between 5 to 10 for GF-based algorithm, and it is fixed at 5 for combo algorithm. This means the coreset size of the whole window M is 155 for combo algorithm and 155 to 310 for GF-based algorithm. We use value of k as 4 for this experiment.

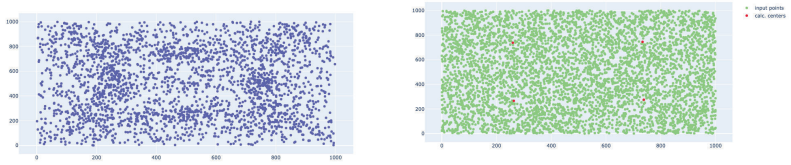


(a) Cost vs. Window number

(b) Coreset Size vs. Window number

Figure 32 Cost and Coreset Size variation with Window number

Results from GF-based algorithm:

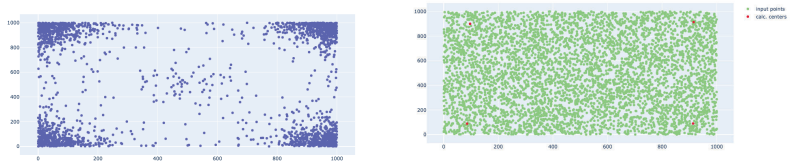


(a) Coreset Points

(b) Input points with centers generated from Coreset

Figure 33 Coreset made by GF-based Algorithm and centers generated by k-means++

Results from Combo Algorithm:



(a) Coreset Points

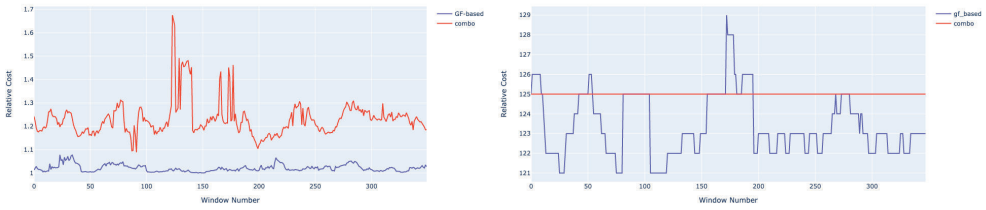
(b) Input points with centers generated from Coreset

Figure 34 Coreset made by Combo Algorithm and centers generated by k-means++

As we see in Figure 33a and Figure 34a combo algorithm seem to be finding coreset points at the corners of the point distribution whereas GF-based algorithm finds a well distributed coreset. Thus, the coreset generated by GF-based algorithm is a much better representation of the input distribution compared to the coreset generated by combo algorithm. In Figure 33b, we see that the k centers are uniformly distributed within the box. In Figure 34b, we see that the k centers are at the corners of the distribution. This increases the average cost of combo algorithm as is seen in Figure 32a.

6.4 Comparing on S1 Dataset

For this experiment, a window size of 1000 was used. θ was set to 2. L was set to 40 and the coreset size per pane varies between 5 to 10 for GF-based algorithm, and it is fixed at 5 for combo algorithm. This means the coreset size of the whole window M is 125 for combo algorithm and 125 to 250 for GF-based algorithm. We set the value of $k = 3$.



(a) Cost vs. Window number

(b) Coreset Size vs. Window number

Figure 35 Cost and Coreset Size variation with Window number

We notice in Figure 35b that at some instances the plot shows a drop in the coreset size of GF-based algorithm below 125. This can happen when L points are seen before m coreset points are formed in a pane. This happens when the points lie very close to one another and a large number of points get absorbed in the same GF.

Results from GF-based Algorithm:

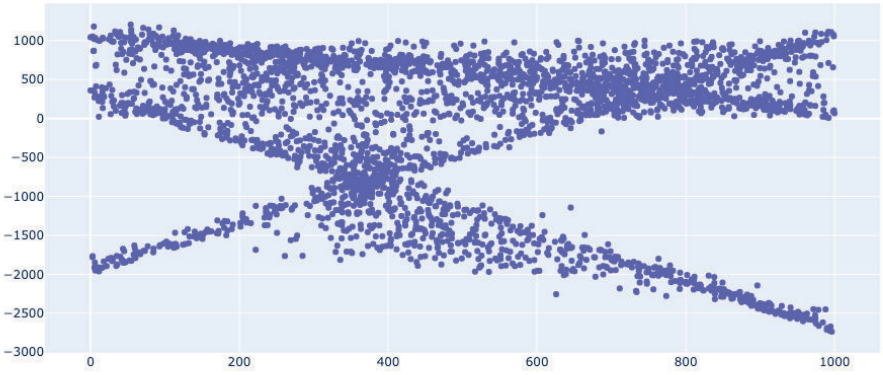


Figure 36 Coreset made by GF-based Algorithm

Results from Combo Algorithm:

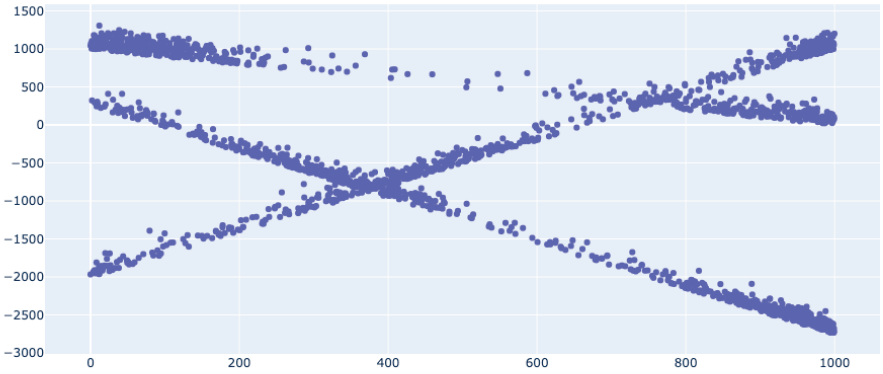


Figure 37 Coreset made by Combo Algorithm

Figure 36 and Figure 37 shows the coresets created by GF-based algorithm and combo algorithm respectively. Here too, we see, GF-based algorithm finds coresets outside the cluster of input points. We also notice that, combo algorithm finds coreset points on the outskirts of the input dataset. Surprisingly, we notice from Figure 35a that combo algorithm has a higher average cost than GF-based algorithm. We are not exactly sure why this happens but we assume the reasoning was same as was given in Section 6.1.

Comparing the Differences and Similarities in the Outcome between the Different Sections

In all the above experiments, we notice that combo algorithm has a tendency to find coreset points on the periphery of the input distribution. Moreover, GF-based algorithm tends to join the centers of the clusters of gaussian distributed datasets. If the clusters are far way, such behaviour of GF-based algorithm seriously detoriorates the quality of clustering because many coreset points generate far away from the actual input cluster. In such scenarios, combo algorithm performs better. But for datasets where points lie close to each other, GF-based algorithm always performs better than combo algorithm.

7 Conclusion

In this thesis, we studied Ackermann's algorithm [1] and Youn's GF-based algorithm [22] extensively. Then we studied the effect of different parameters on GF-based algorithm's performance and suggest on how this values should be chosen for two-dimensional gaussian datasets.

We further extended Ackermann's algorithm to the sliding window setting using ideas from Youn's algorithm. For the new algorithm, instead of the GF-creation method of Youn, we use the Coreset Tree technique in each pane to handle incoming points in a stream. We named this the Combo Algorithm. We ran experiments on this new algorithm and studied the impact of its parameters: bucket size and slide length on the quality of clustering.

Subsequently, our goal was to compare the performance of the two algorithms in terms of clustering quality and space usage by running them on datasets with different distributions. We concluded that for far separated gaussian clusters, combo algorithm performs better than the GF-based algorithm although for closely lying clusters, the results are opposite. We also noticed that combo algorithm has a tendency to find coreset points on the periphery of the input distribution whereas, GF-based algorithm tends to find coreset points on the line joining two clusters. We also noticed that GF-based algorithm performed better for uniform distribution of input points. Furthermore, GF-based algorithm also performed better for input points distributed along a line.

Further investigation can be done using a bigger bucket size for the combo algorithm and a bigger slide length for GF-based algorithm while maintaining the coreset size comparable. Also, new experiments can be performed on higher-dimensional and real-world datasets to evaluate the two algorithms.

References

1 Marcel R. Ackermann, Marcus Märtens, Christoph Raupach, Kamil Swierkot, Christiane Lammersen, and Christian Sohler. StreamKM++: A clustering algorithm for data streams. *ACM Journal of Experimental Algorithmics*, 17, July 2012. URL: https://dl.acm.org/doi/10.1145/2133803.2184450, doi:10.1145/2133803.2184450.

2 Charu C. Aggarwal, Philip S. Yu, Jiawei Han, and Jianyong Wang. - A Framework for Clustering Evolving Data Streams. In Johann-Christoph Freytag, Peter Lockemann, Serge Abiteboul, Michael Carey, Patricia Selinger, and Andreas Heuer, editors, *Proceedings 2003 VLDB Conference*, pages 81–92. Morgan Kaufmann, San Francisco, January 2003. URL: https://www.sciencedirect.com/science/article/pii/B9780127224428500161, doi:10.1016/B978-012722442-8/50016-1.

3 David Arthur and Sergei Vassilvitskii. k-means++: the advantages of careful seeding. In Nikhil Bansal, Kirk Pruhs, and Clifford Stein, editors, *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2007, New Orleans, Louisiana, USA, January 7-9, 2007*, pages 1027–1035. SIAM, 2007. URL: http://dl.acm.org/citation.cfm?id=1283383.1283494.

4 Michele Borassi, Alessandro Epasto, Silvio Lattanzi, Sergei Vassilvitskii, and Morteza Zadimoghaddam. Sliding Window Algorithms for k-Clustering Problems. *arXiv:2006.05850 [cs]*, October 2020. arXiv: 2006.05850. URL: http://arxiv.org/abs/2006.05850.

5 V. Braverman and R. Ostrovsky. Smooth Histograms for Sliding Windows. In *48th Annual IEEE Symposium on Foundations of Computer Science (FOCS'07)*, pages 283–293, October 2007. ISSN: 0272-5428. doi:10.1109/FOCS.2007.55.

6 Vladimir Braverman, Harry Lang, Keith Levin, and Morteza Monemizadeh. Clustering on Sliding Windows in Polylogarithmic Space. page 15 pages, 2015. Artwork Size: 15 pages Medium: application/pdf Publisher: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik GmbH, Wadern/Saarbruecken, Germany. URL: http://drops.dagstuhl.de/opus/volltexte/2015/5654/, doi:10.4230/LIPICS.FSTTCS.2015.350.

7 Vladimir Braverman, Rafail Ostrovsky, and Carlo Zaniolo. Optimal sampling from sliding windows. In *Proceedings of the twenty-eighth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '09, pages 147–156, New York, NY, USA, June 2009. Association for Computing Machinery. doi:10.1145/1559795.1559818.

8 Matteo Ceccarello, Andrea Pietracaprina, and Geppino Pucci. Solving k-center Clustering (with Outliers) in MapReduce and Streaming, almost as Accurately as Sequentially. *arXiv:1802.09205 [cs]*, January 2019. arXiv: 1802.09205. URL: http://arxiv.org/abs/1802.09205.

9 Moses Charikar, Sudipto Guha, Éva Tardos, and David B. Shmoys. A Constant-Factor Approximation Algorithm for the k-Median Problem. *Journal of Computer and System Sciences*, 65(1):129–149, August 2002. URL: https://www.sciencedirect.com/science/article/pii/S0022000002918829, doi:10.1006/jcss.2002.1882.

10 Vincent Cohen-Addad, Chris Schwiegelshohn, and Christian Sohler. Diameter and k-center in sliding windows. In Ioannis Chatzigiannakis, Michael Mitzenmacher, Yuval Rabani, and Davide Sangiorgi, editors, *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016, July 11-15, 2016, Rome, Italy*, volume 55 of *LIPIcs*, pages 19:1–19:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.ICALP.2016.19.

11 Vincent Cohen-Addad, Chris Schwiegelshohn, and Christian Sohler. Diameter and k-Center in Sliding Windows. In Ioannis Chatzigiannakis, Michael Mitzenmacher, Yuval Rabani, and Davide Sangiorgi, editors, *43rd International Colloquium on Automata, Languages, and Programming (ICALP 2016)*, volume 55 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 19:1–19:12, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISSN: 1868-8969. URL: http://drops.dagstuhl.de/opus/volltexte/2016/6340, doi:10.4230/LIPIcs.ICALP.2016.19.

12 Mayur Datar and Rajeev Motwani. The Sliding-Window Computation Model and Results. In Charu C. Aggarwal, editor, *Data Streams: Models and Algorithms*, pages 149–167. Springer US, Boston, MA, 2007. doi:10.1007/978-0-387-47534-9_8.

13 Teofilo F. Gonzalez. Clustering to minimize the maximum intercluster distance. *Theor. Comput. Sci.*, 38:293–306, 1985. doi:10.1016/0304-3975(85)90224-5.

14 Gramoz Goranci, Monika Henzinger, Dariusz Leniowski, Christian Schulz, and Alexander Svozil. Fully dynamic k-center clustering in low dimensional metrics. In Martin Farach-Colton and Sabine Storandt, editors, *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2021, Virtual Conference, January 10-11, 2021*, pages 143–153. SIAM, 2021. doi:10.1137/1.9781611976472.11.

15 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(03):515–528, May 2003. Publisher: IEEE Computer Society. URL: https://www.computer.org/csdl/journal/tk/2003/03/k0515/13rRUIIVlkB, doi:10.1109/TKDE.2003.1198387.

16 Sariel Har-Peled and Akash Kushal. Smaller Coresets for k-Median and k-Means Clustering. *Discrete & Computational Geometry*, 37(1):3–19, January 2007. doi:10.1007/s00454-006-1271-x.

17 Sariel Har-Peled and Soham Mazumdar. On coresets for k-means and k-median clustering. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing - STOC '04*, page 291, Chicago, IL, USA, 2004. ACM Press. URL: http://portal.acm.org/citation.cfm?doid=1007352.1007400, doi:10.1145/1007352.1007400.

18 Dorit S. Hochbaum and David B. Shmoys. A best possible heuristic for the k-center problem. *Math. Oper. Res.*, 10(2):180–184, 1985. doi:10.1287/moor.10.2.180.

19 Stuart P. Lloyd. Least squares quantization in PCM. *IEEE Trans. Inf. Theory*, 28(2):129–136, 1982. doi:10.1109/TIT.1982.1056489.

20 James MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297. Oakland, CA, USA, 1967.

21 Richard Matthew McCutchen and Samir Khuller. Streaming algorithms for k-center clustering with outliers and with anonymity. In Ashish Goel, Klaus Jansen, José D. P. Rolim, and Ronitt Rubinfeld, editors, *Approximation, Randomization and Combinatorial Optimization. Algorithms and Techniques, 11th International Workshop, APPROX 2008, and 12th International Workshop, RANDOM 2008, Boston, MA, USA, August 25-27, 2008. Proceedings*, volume 5171 of *Lecture Notes in Computer Science*, pages 165–178. Springer, 2008. doi:10.1007/978-3-540-85363-3_14.

22 J. Youn, J. Shim, and S. Lee. Efficient Data Stream Clustering With Sliding Windows Based on Locality-Sensitive Hashing. *IEEE Access*, 6:63757–63776, 2018. Conference Name: IEEE Access. doi:10.1109/ACCESS.2018.2877138.

23 Tian Zhang, Raghu Ramakrishnan, and Miron Livny. BIRCH: an efficient data clustering method for very large databases. In H. V. Jagadish and Inderpal Singh Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, pages 103–114. ACM Press, 1996. doi:10.1145/233269.233324.

24 Y. Zhang, K. Tangwongsan, and S. Tirthapura. Streaming k-Means Clustering with Fast Queries. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 449–460, April 2017. ISSN: 2375-026X. doi:10.1109/ICDE.2017.102.