StreamXM: An Adaptive Partitional Clustering Solution for Evolving Data Streams

Robert Anderson and Yun Sing Koh(✉)

Department of Computer Science, University of Auckland, Auckland, New Zealand
rand079@aucklanduni.ac.nz, ykoh@cs.auckland.ac.nz

Abstract. A challenge imposed by continuously arriving data streams is to analyze them and to modify the models that explain them as new data arrives. We propose StreamXM, a stream clustering technique that does not require an arbitrary selection of number of clusters, repeated and expensive heuristics or in-depth prior knowledge of the data to create an informed clustering that relates to the data. It allows a clustering that can adapt its number of classes to those present in the underlying distribution. In this paper, we propose two different variants of StreamXM and compare them against a current, state-of-the-art technique, StreamKM. We evaluate our proposed techniques using both synthetic and real world datasets. From our results, we show StreamXM and StreamKM run in similar time and with similar accuracy when running with similar numbers of clusters. We show our algorithms can provide superior stream clustering if true clusters are not known or if emerging or disappearing concepts will exist within the data stream.

Keywords: StreamXM · StreamKM · X-means · Clustering · Streaming · Adaptive · Data-streams · Unsupervised learning

1 Introduction

Data stream mining has arisen as a key technique in data analysis. As detailed by Gaben et al. [1], data streams may continually generate new information, with no upper bound on the information produced by the stream. With data streams, clustering has a broad range of real-world applications. For instance, it has been found to be effective for differentiating Twitter posts by bots and real people [2], detecting fraudulent credit-card transactions [3] and detecting malicious traffic over networks [4]. In all of these cases, running the analysis in real-time provides extra value: bots can be screened from Twitter feeds, fraudulent credit card transactions can be denied and malicious traffic can be rejected, with hosts banned. In all of these cases, clustering data-streams allows potential value that traditional approaches to analysis would not.

Currently, there are few partitional clustering techniques for data stream mining that can adapt to unseen concepts as they emerge in a data stream. Partitional clustering creates a simple and effective way to describe datasets

S. Madria and T. Hara (Eds.): DaWaK 2015, LNCS 9263, pp. 270–282, 2015.
DOI: 10.1007/978-3-319-22729-0_21

that can provide an interpretable explanation of how data is changing over the life of a dataset, making it a useful approach to clustering for our problem. However, most partitional clustering approaches in a data stream require a fixed number of clusters to be predefined that cannot change over the lifespan of the stream. This will not be an effective technique for analysing data which changes over time. In this research, we proposed a partitional clustering technique that can adapt to underlying concepts without being constrained by a fixed level of clustering.

We introduce the novel idea of combining an approximation-based data-stream clustering approach with a clustering approach that uses the Bayes Information Criterion, a penalised-likelihood measure, to create an effective clustering for a given window of data. We propose two variants of our technique: StreamXM, and StreamXM with Lloyds, which relies upon Lloyds algorithm. Both these techniques are evolving partition-based clustering techniques for data streams. By adapting previous work in the area StreamKM's coreset construction [5] and X-means analyses upon these coresets, we combine the benefit of an efficient and cheap algorithm with the flexibility of another.

Two major contributions of our research include: (1) proposing implementations that can achieve similar quality of the less-flexible StreamKM's clustering on static data while retaining levels of memory usage and runtime that are acceptable in a streaming environment. (2) providing a partitional clustering technique that can adapt to appearing or disappearing concepts in underlying data, thus achieving similar or better quality and performance of clustering than less-flexible techniques without requiring a specific level of clustering to use.

In our next section, we discuss the research work in the area. In Sect. 3 we discuss our proposed algorithm. In Sect. 4, we briefly discuss our experimental setup and the datasets to be used for our comparative analysis. In the following section, we explore the significance of our results. Finally, we conclude the paper in Sect. 6.

2 Related Work

In recent years there has been a broad range of research in the area of clustering data streams [6–8]. A highly-regarded and well-known clustering technique that utilises hierarchical clustering is BIRCH [9]. This algorithm takes n instances and a desired number of clusters, and calculates the clustering feature of the set, comprising of the linear sum and square sum of instances. This method is shown to run quickly, but does not minimise sum-of-squared error well when compared to other current data stream clustering methods [5].

Guha et al. [10] proposed a method based upon a k-median approach, entitled STREAM. A k-median approach is similar to k-means, except that the value selected for a cluster centre must be the median value for instances within the cluster grouping. This clusterer reduces the sum-of-squared error when compared to a clusterer like BIRCH, but runs in superlinear time.

StreamKM [5] proposed a clustering algorithm that would allow k-means++ to achieve speeds required for data-stream clustering. They used a merge-and-reduce technique to reduce a large set of data points to a representative coreset tree. This allows fast seeding for k-means++ and by using a coreset size significantly smaller than the total dataset size, they could justify running complex operations such as k-means++ upon it. They explore the adaptive coreset, which provides a weighted multiset as a basis for the seeding. Using sampling based upon merge-and-reduce, they could maintain a sample of a fixed size that could represent all data seen, with a greater weighting placed on recent data. Use of this technique has been supported by recent applications for real-world data. For example, Miller et al. [2] use a variant of StreamKM for a proposed Twitter spammer detector that performed very accurately.

3 The StreamXM Algorithms

Here we detail our proposed algorithms, (1) StreamXM with Lloyds and (2) StreamXM, to allow adaptive clustering of unseen data.

Both of our proposed algorithms use X-Means [11], an approach that takes a minimum and maximum clustering size, and measures quality of clusterings in that range by the Bayes Information Criterion. This is a penalised-likelihood approach that weighs the explanatory value of more clusters against the increased complexity of the model. It considers the relative improvement in BIC by splitting each cluster. To measure this, it runs local k-means within each cluster, if it were to be split into two new clusters, and calculates whether these child clusters better explain the data than their parent. This allows an informed approach to choosing a sensible clustering without prior knowledge. Both of our technique use X-means in a slightly different manner.

3.1 StreamXM with Lloyds

The first implementation of StreamXM that we propose combines the strength of StreamKM's implementation of the Lloyd's algorithm with the adaptability of X-Means. The Lloyd's algorithm [12] is a local search that seeks to iteratively minimise the sum of squares after initialising with random centres, until reaching a stable level. This is common to many implementations of k-means. However, traditional k-means implementations often only guarantee finding local optima. It utilises D^2-weighting, encouraging centres to maximise their distance from each other on initialisation, and shows that it leads to significant speed improvements over traditional k-means clustering. We initially run X-means to determine the correct number of cluster and the centroids. Using the information derived from X-means, we then run k-means on static data for the minimum number of clusters set. Essentially, in our first iteration, it uses X-Means to determine the clustering level k for a coreset, and then uses the powerful implementation of k-means++ used by StreamKM to cluster the coreset.

The technique takes a stream of data instances and uses the merge-and-reduce technique to construct a coreset tree as it receives them, with the aim of keeping the entire data stream summarised within $\lceil log_2(n/m) + 2 \rceil$ buckets. A full bucket represents $2^{i-1}m$ instances sent to the algorithm, with i varying by the total number of instances inserted in the coreset tree [5]. Once the width is reached, an X-means clustering of the coreset is performed. It should be noted that in our implementation, and as per the initial X-means algorithm, the weight of each coreset point is disregarded. The coreset points that exist should provide a sufficient approximation of the source data without weights, especially once the data stream has been running for some time.

The algorithm we propose can be seen as working as follows. First, it compresses a set interval of streamed instances into a coreset, similarly to StreamKM. It then runs X-Means on the interval, resulting in a clustering of a set number of clusters. It checks the sum of squared errors between the cluster centres their points, and tries running k-means++, using Lloyds algorithm, five times (as found as a good combination of speed and quality by Ackermann et al. [5]), seeking a clustering with the same number of centres as the X-Means run, but with a reduced sum of squared error. It compares the X-Means clustering and five k-means++ clusterings and chooses that with the lowest sum of squared error. The process then repeats for each further interval.

Algorithm 1. StreamXM with Lloyds

```
1: procedure StreamXM with Lloyds
2:     for each data point:
3:     Insert data point into coreset tree
4:     if pointCounter % width == 0 then
5:         xmeansClustering ← X-means(coreset, minClusters, maxClusters)
6:         numCentres ← xmeansClustering.numClusters()
7:         curCost ← xmeansClustering.sumOfSquares()
8:         clusterCentres ← xmeansClustering.centres()
9:         repeat 5 times:
10:        kmeansPlusPlusClustering ← k-means++(coreset, numCentres)
11:        if kmeansPlusPlusClustering.cost() < curCost then
12:            curCost ← kmeansPlusPlusClustering.cost()
13:            clusterCentres ← kmeansPlusPlusClustering.centres()
14:    return clusterCentres
```

X-means will return the optimal clustering it can find for the coreset given different levels of k between **minClusters** and **maxClusters** [11]. It performs an initial k-means split with $k = 2$ upon the data. For the resulting two clusters, it considers splitting each cluster through running k-means with $k = 2$ on individual clusters. It calculates the relative Bayes Information Criterion (BIC) change for the optimal single-cluster split, and considers how well this model explains the coreset data when compared to the simpler model. The BIC of Pelleg and Moore's model penalises the log-likelihood of each model against its complexity, when deciding which to use.

Once this is completed, StreamXM repeats k-means++ on the data five times (as it is the number of iterations that the authors of StreamKM found worked best for their algorithm), with a k value decided by X-means. It will select the optimal clustering based upon sum of squared distance. This ensures that our algorithm provides the adaptivity of repeated X-means runs while gaining the benefit of repeated efficient k-means runs upon the coreset. This algorithm shares StreamKM's parameters of coreset size and width, and X-Means's parameters of minimum and maximum clusters to use.

We can infer the worst case time complexity of the algorithm through considering the algorithms it is based upon. We have inherited the coreset operations from StreamKM, as well as the repeated runs of k-means++. In [5], the authors show that by performing its complex operations (such as k-means++ and merge-and-reduce steps upon the coreset tree) upon the coreset of fixed size m rather than the total dataset, StreamKM avoids super-linear complexity. They show the algorithm performs in $\mathcal{O}(dnm)$ time, with d referring to the dimensionality of the data.

The BIC calculation can be performed in $\mathcal{O}(k)$ time and could be run a maximum $1 + k(k + 1)$ times per X-means, where k represents **maxClusters**. Traditional k-means runs in $\mathcal{O}(knd)$ time, but X-means only ever runs k-means for $k = 2$ for any run, though runs k-means $\frac{k(k+1)}{2}$ times. The implementation uses kd-trees, which avoid recalculating distance for subsets of points joined to a centroid through a process called 'blacklisting' described in [13], which leads to an average time-complexity that is often sub-linear [11]. X-means is only being run on the coreset, and not n. None of the factors that affect running time of X-means change as n changes, so we can state that the total additional time to run X-means scales linearly with n. We must select a coreset size m, a dimensionality d and a suitably restrained **maxClusters** value so that the increased time fits within the constraints of our application. Model memory requirements of StreamKM are shown to be $\Theta(dm \log(n/m))$ in [5], and our model introduces no further memory requirements beyond the coreset tree stored in StreamKM.

3.2 StreamXM

The second implementation of StreamXM we propose abandons k-means++ and relies on clusterings delivered by X-Means. Our technique takes an interval of data and constructs a coreset to represent it with a reduced number of points. It then runs X-Means upon the data, finding a level of k that is appropriate for the interval, as well as coordinates of k cluster centres. It then repeats X-Means on the same set of data $xmIter - 1$ times. Each time it reruns X-Means, it considers whether the new clustering it finds has a reduced sum-of-squared errors when compared to the previous clusterings it found. It selects the clustering with the lowest sum-of-squared errors as the chosen clustering for that interval. This algorithm is run on every interval of data until the stream ends. The pseudocode is shown in Algorithm 2.

Algorithm 2. StreamXM

```
1: procedure StreamXM
2:     for each data point:
3:     Insert data point into coreset tree
4:     if pointCounter % width == 0 then
5:         xmeansClustering ← X-means(coreset, minClusters, maxClusters)
6:         numCentres ← xmeansClustering.numClusters()
7:         curCost ← xmeansClustering.sumOfSquares()
8:         clusterCentres ← xmeansClustering.centres()
9:         repeat xmIter − 1 times:
10:        newXMeansClustering ← X-means(coreset, minClusters, maxClusters)
11:        if newXMeansClustering.cost() < curCost then
12:            curCost ← newXMeansClustering.cost()
13:            clusterCentres ← newXMeansClustering.centres()
14:    return clusterCentres
```

This algorithm has the same parameters as StreamXM with Lloyds. It also has *xmIter*, the number of times X-Means is to be run on each interval of data. This allows StreamXM multiple chances to find a better clustering for the coreset, and decides on the best through a minimised sum-of-squares. Therefore, an increased value of *xmIter* will tend to lead to a higher average clustering, and will also take more time. As described in the Experimental Setup section, we test to find the ideal value of *xmIter*, which allows a quality clustering without severely impacting runtime. The memory requirements of this algorithm will be the same as StreamXM with Lloyds.

4 Experimental Setup

To test our implementation, we used the MOA API (available from http://moa.cms.waikato.ac.nz), which provided an implementation of StreamKM to compare against. All experiments were run on an Intel Core i7-3770S running at 3.10GHz with 16.0GB of RAM on a 64-bit Windows 7.

Across our experiments, we used several evaluation measures to explore our algorithms' accuracy, speed and memory use. For every set of tests run, the experiment was repeated 30 times per set of factors, so as to receive a suitably representative mean and standard deviation for the underlying distribution.

To compare the quality of our clusterings we used two measures. First, we measured the sum of squared errors (SSQ) between cluster centres and actual instances belonging to each cluster centre across the experiment. This was calculated for each instance according to the clustering at the end of the interval that the instance was present within. As per Eq. 1, the SSQ is the sum across all clusters $i = 1$ to $i = K$ of all instances in each cluster x of the squared Euclidean distance between the instance and the cluster centre m_i. The sum of SSQ across all intervals gave the total SSQ for an experiment.

$$SSQ_{interval} = \sum_{i=1}^{K} \sum_{x \in C_i} dist^2(m_i, x) \tag{1}$$

Second, we measured purity of resulting clusterings. We calculated purity as per Eq. 2. ω represents the set of clusters in each interval-clustering while $\mathbb{C}$ represents the set of classes. We take the sum of the maximum class j within each cluster k across all intervals. Divided by the total number of instances N, we retrieve the purity for an experiment [14].

$$Purity(\Omega, \mathbb{C}) = \frac{1}{N} \sum_{k} \max_{j} |\omega_k \cap c_j| \tag{2}$$

4.1 Datasets

Our goal in testing our algorithms was: to show that they can perform comparably well in a static environment with StreamKM; to show that they can adapt to an underlying change in the number of clusters; and to demonstrate their value when used for analysis of real world data.

For thes synthetic dataset, we decided to use **BIRCHCluster**, a cluster generator included in the WEKA API (http://www.cs.waikato.ac.nz/ml/weka/). The advantage of this tool is that it creates a specified number of instances that fall into clusters in a spherical pattern, with a set dimensionality, with a set radius. BIRCH is primarily known as a clustering method [9], but can also generate clustering problems to test other techniques upon. Using **BIRCHCluster**, we created thirty datasets of differing size (250,000, 500,000 and 1 million instances), clusterings (5, 10, 20 and 50 underlying clusters). To generate an evolving dataset, we first generated the stream largest number of custers needed. To replicate dissapearing clusters, we would remove x number of clusters from the stream pass a point t. Vice versa if we simulated appearing clusters we would remove x number of clusters from the beginning of the stream until a point t. In our experiment t was set as half the entire stream size.

We ran both algorithms on the KDD Cup '99 Network Intrusion dataset. For consistency in our paper, we have split each dataset into 10 intervals for evaluation.

5 Results

In this section, we summarise results found from our experiments. First, we compare StreamXM's to StreamKM across in a static (non-evolving) stream. Second, we compare StreamXM's to StreamKM in an evolving stream. In the experiments, memory use are in bytes and runtime in milliseconds. For every set of tests run, the experiment was repeated 30 times per set of factors.

5.1 Performance and Quality of StreamXM on Static Datasets

These experiments compare our techniques StreamXM (**sxm**) and StreamXM with Lloyds (**sxmlloyds**) against StreamKM run with a value of k: half of the true clustering (**skmhalf**); the same as the true clustering (**skmfull**); and the same as the mean clustering used by StreamXM (**skm**). We have two main aims with these experiments. The first is to show that the quality of clusterings delivered by StreamXM and StreamXM with Lloyds is at least comparable with those delivered by StreamKM across datasets with varied clustering, dimensionality and stream size. The second is to show that the performance of StreamXM and StreamXM with Lloyds is suitable for a stream environment in which the runtime and memory have at worst a linear relationship with the varied factors in each experiment.

We are showing that our StreamXM variants work comparably to StreamKM on static datasets, as data streams when data is not evolving over time. All of these experiments used the parameters decided upon was: $xmIter = 3$ for StreamXM,a coreset of size 1000 and the BIC as the information criterion for StreamXM and StreamXM with Lloyds. $xmIter = 3$ was chosen as extensive experimentation showed that it was a reasonable setting. As these are static datasets, StreamXM variants will gain no advantage from their adaptive nature as there are no changes within the stream.

Here it is important to note that **skmfull** will always produce a close to perfect results for both Purity and SSQ measures as it was given perfect knowledge of the number of clusters available. In a real world data stream, determining the correct number of cluster before the stream has arrived is an impossible task. However for completeness we have included **skmfull** in our experiments.

Performance and Quality of StreamXM with Varying Clusters. Here, our experiments tested purity, sum-of-squared errors, memory usage and runtime in problems with 5, 10, 20 and 50 underlying (true) clusters. Here, we sought to explore how an increasing number of clusters affects the quality and performance of our techniques against StreamKM, and to ensure that increasing the underlying clustering of a dataset does not increase the time taken to cluster it in a superlinear fashion. We ran experiments with datasets of 5 dimensions and 250,000 instances.

Table 1 shows the performance of StreamXM and StreamXM with Lloyds in comparison with StreamKM. When we ran **skmfull** it was always be the best performer in a static environment because it is set to run with the true number of clusters in the underlying data stream. In reality it is highly improbable for the clusterer to be perfectly set for an unseen data stream, thus not reported in the table, due to space restrictions. In Table 2 we show the runtime and memory performance of our techniques. Although there is a slight overhead in our technique, we show in the next set of experiments that this is not an exponential problem as the stream size increases. Memory use across all techniques is consistently similar, and does not increase significantly with increases in the true clustering level. We can surmise that increased levels of k used in a model do not contribute to an increase in model memory requirements.

Table 1. Quality measures and clustering for algorithms run with varied true clustering

Model	Purity ± 95%CI	SSQ $\times 10^3$ ± 95%CI	Mean k ± 95%CI
True clustering = 5 and minClusters = 3			
skm	0.979 ± 0.020	2.277 ± 2.057	4.867 ± 0.124
skmhalf	0.681 ± 0.014	39.209 ± 3.857	3.000 ± 0.000
sxm	0.964 ± 0.011	4.798 ± 1.598	4.815 ± 0.085
sxmlloyds	0.898 ± 0.021	10.662 ± 2.951	4.347 ± 0.118
True clustering = 10 and minClusters = 5			
skm	0.857 ± 0.016	6.030 ± 0.906	8.100 ± 0.172
skmhalf	0.583 ± 0.010	28.323 ± 2.202	5.000 ± 0.000
sxm	0.834 ± 0.013	12.245 ± 1.446	8.037 ± 0.135
sxmlloyds	0.795 ± 0.012	10.589 ± 1.212	7.371 ± 0.129
True clustering = 20 and minClusters = 10			
skm	0.802 ± 0.013	5.575 ± 0.635	14.967 ± 0.239
skmhalf	0.580 ± 0.007	17.431 ± 1.063	10.000 ± 0.000
sxm	0.769 ± 0.011	11.176 ± 0.945	14.942 ± 0.192
sxmlloyds	0.765 ± 0.011	7.402 ± 0.704	14.145 ± 0.198
True clustering = 50 and minClusters = 25			
skm	0.750 ± 0.006	4.702 ± 0.282	34.467 ± 0.278
skmhalf	0.578 ± 0.005	10.807 ± 0.393	25.000 ± 0.000
sxm	0.684 ± 0.009	10.325 ± 0.559	34.459 ± 0.303
sxmlloyds	0.728 ± 0.007	5.536 ± 0.311	33.202 ± 0.274

Table 2. Performance measures for algorithms run with varied true clustering

Model	Runtime ± 95%CI	Memory $\times 10^6$ ± 95%CI
True clustering = 5 and minClusters = 3		
skm	2912 ± 103	1.211 ± 0.019
skmhalf	2865 ± 99	1.207 ± 0.017
sxm	3115 ± 111	1.208 ± 0.018
sxmlloyds	3004 ± 108	1.209 ± 0.018
True clustering = 10 and minClusters = 5		
skm	2900 ± 91	1.211 ± 0.023
skmhalf	2822 ± 85	1.210 ± 0.022
sxm	3161 ± 102	1.210 ± 0.023
sxmlloyds	3016 ± 95	1.211 ± 0.024
True clustering = 20 and minClusters = 10		
skm	3083 ± 86	1.204 ± 0.021
skmhalf	2976 ± 82	1.205 ± 0.020
sxm	3626 ± 96	1.202 ± 0.020
sxmlloyds	3309 ± 91	1.204 ± 0.020
True clustering = 50 and minClusters = 25		
skm	3688 ± 70	1.222 ± 0.017
skmhalf	3528 ± 65	1.218 ± 0.018
sxm	5046 ± 89	1.214 ± 0.018
sxmlloyds	4466 ± 72	1.219 ± 0.017

Table 3. Quality measures and clustering for algorithms run with varied stream size

Model	Purity ± 95%CI	SSQ $\times 10^3$ ± 95%CI	Mean k ± 95%CI
Stream size = approx.250,000			
skm	0.857 ± 0.016	6.030 ± 0.906	8.100 ± 0.172
skmhalf	0.583 ± 0.009	28.323 ± 2.202	5.000 ± 0.000
sxm	0.834 ± 0.013	12.246 ± 1.446	8.037 ± 0.135
sxmlloyds	0.795 ± 0.012	10.590 ± 1.212	7.371 ± 0.129
Stream size = approx.500,000			
skm	0.855 ± 0.016	12.547 ± 1.859	8.067 ± 0.161
skmhalf	0.582 ± 0.010	56.572 ± 4.519	5.000 ± 0.000
sxm	0.833 ± 0.013	24.127 ± 2.773	8.022 ± 0.117
sxmlloyds	0.799 ± 0.012	20.618 ± 2.450	7.431 ± 0.130
Stream size = approx.1,000,000			
skm	0.848 ± 0.013	25.781 ± 3.425	8.000 ± 0.133
skmhalf	0.582 ± 0.009	112.763 ± 8.920	5.000 ± 0.000
sxm	0.833 ± 0.012	48.202 ± 5.206	8.019 ± 0.110
sxmlloyds	0.796 ± 0.011	42.015 ± 4.669	7.389 ± 0.118

Table 4. Performance measures for algorithms run with varied stream size

Model	Runtime ± 95%CI	Memory $\times 10^6$ ± 95%CI
Stream size = approx. 250,000		
skm	2880 ± 89	1.211 ± 0.023
skmhalf	2799 ± 85	1.210 ± 0.022
sxm	3141 ± 99	1.211 ± 0.023
sxmlloyds	2989 ± 93	1.211 ± 0.024
Stream size = approx. 500,000		
skm	5818 ± 194	1.201 ± 0.022
skmhalf	5656 ± 186	1.197 ± 0.020
sxm	6317 ± 201	1.201 ± 0.022
sxmlloyds	6016 ± 195	1.196 ± 0.021
Stream size = approx. 1,000,000		
skm	11659 ± 359	1.216 ± 0.020
skmhalf	11378 ± 352	1.214 ± 0.019
sxm	12778 ± 392	1.214 ± 0.019
sxmlloyds	12136 ± 394	1.211 ± 0.018

Performance and Quality of StreamXM with Varying Stream Size. Our experiments tested purity, sum-of-squared errors, memory usage and runtime in problems with 250,000, 500,000 and 1 million data iteams within the dataset. We explore how an increasing stream size affects the quality and performance of StreamXM against StreamKM. We ran experiments with datasets of 10 underlying clusters and 5 dimensions. In Table 3, we see sum-of-squared error scaling linearly with the number of instances across all models. Purity holds constant across all models. There is no evidence of StreamXM or StreamXM with Lloyds altering their mean k based upon the stream size.

In Table 4, we can see evidence of runtime and memory use scaling linearly across all models with the total number of instances in our data stream, with time increasing between 99.7 % and 102.4 % when doubling the number of instances for all models.

Table 5. Quality measures and clustering for algorithms run on evolving data streams

Model	Purity ± 95%CI	SSQ $\times 10^3$ ± 95%CI	Mean k ± 95%CI
Stream shrinking from 6 clusters to 3 clusters			
skm	1.000 ± 0.000	0.408 ± 0.002	6.000 ± 0.000
sxm	0.942 ± 0.008	23.469 ± 4.549	4.564 ± 0.116
sxmlloyds	0.913 ± 0.010	29.630 ± 4.666	4.146 ± 0.128
Stream growing from 3 clusters to 6 clusters			
skm	0.797 ± 0.008	90.654 ± 7.391	3.000 ± 0.000
sxm	0.942 ± 0.008	22.831 ± 3.931	4.560 ± 0.122
sxmlloyds	0.911 ± 0.010	31.056 ± 4.804	4.159 ± 0.134

Table 6. Performance measures for algorithms run on evolving data streams

Model	Runtime ± 95%CI	Memory $\times 10^6$ ± 95%CI
Stream shrinking from 6 clusters to 3 clusters		
skm	13504 ± 478	1.228 ± 0.018
sxm	12841 ± 442	1.227 ± 0.018
sxmlloyds	12561 ± 450	1.226 ± 0.018
Stream growing from 3 clusters to 6 clusters		
skm	11180 ± 360	1.229 ± 0.016
sxm	12424 ± 395	1.229 ± 0.017
sxmlloyds	12155 ± 415	1.230 ± 0.016

5.2 Performance and Quality of StreamXM Run on Evolving Dataset Streams

In these experiments, we ran three algorithms: StreamXM (**sxm**), StreamXM with Lloyds (**sxmlloyds**) and StreamKM (**skm**). For StreamKM, we set k to the clustering level of the evolving dataset in the first section of data, as the change in clustering would not be known about when setting up the algorithm intially.

Examining Table 5, for the shrinking data stream, we see that StreamKM creates a very effective clustering through using $k = 6$ throughout the experiment. StreamXM delivers a very effective clustering with a mean k very close to what we should hope for (almost perfectly halfway between 3 and 6). For the growing data stream, StreamKM fails to create as effective a clustering, as three cluster centres cannot adequately capture the six clusters the data falls around in the second section of data. Both StreamXM and StreamXM with Lloyds maintain the quality of their clusterings achieved with the shrinking dataset. This situation helps highlight the value an adaptive clusterer can add when underlying concepts may change throughout the course of a stream.

From Table 6 StreamKM has a significantly slower runtime than StreamXM with Lloyds for the shrinking data stream, as it takes more time to fit six cluster centres to the data than StreamXM with Lloyds. StreamXM appears faster than StreamKM, but not significantly so in this experiment. When StreamKM has fewer clusters, it is significantly faster than both variants of StreamXM. The runtime for either variant of StreamXM is not significantly different from the other. In Table 7, we examine the change is clustering over each section of the data. As StreamXM and StreamXM with Lloyds are adaptive, we expect a significant change in clusters when the underlying true clustering changes.

5.3 Performance and Quality of StreamXM on Real-World Dataset

In this section, we examine quality and performance of running our techniques on the real-world datasets we have selected. We contrast how well they have done, and compare the overall clustering to the real data. However, we examine the fit of the clusters chosen by StreamXM variants to the data much more closely later in this section. Table 8 shows the relative quality of each clustering

Table 7. Mean clustering for each section of evolving data streams

Model	Actual clusters	Mean k	± 95 %CI
Stream shrinking from 6 clusters to 3 clusters			
skm	6 clusters	6.000	± 0.000
	3 clusters	6.000	± 0.000
sxm	6 clusters	5.080	± 0.033
	3 clusters	4.041	± 0.033
sxmlloyds	6 clusters	4.593	± 0.040
	3 clusters	3.746	± 0.038
Stream growing from 3 clusters to 6 clusters			
skm	6 clusters	3.000	± 0.000
	3 clusters	3.000	± 0.000
sxm	3 clusters	4.040	± 0.034
	6 clusters	5.073	± 0.033
sxmlloyds	3 clusters	3.729	± 0.038
	6 clusters	4.535	± 0.042

Table 8. Quality measures and clustering: Intrusion dataset

Model	Purity	± 95 %CI	SSQ $\times 10^3$	± 95 %CI	Mean k	± 95 %CI
skm	0.574	± 0.004	1.386×10^8	$\pm 0.435 \times 10^8$	5.033	± 0.199
skmfull	0.913	± 0.012	212.224	± 0.017	23.000	± 0.000
skmhalf	0.665	± 0.007	806.396	± 0.010	12.000	± 0.000
sxm	0.966	± 0.009	114.262	± 0.012	5.035	± 0.152
sxmlloyds	0.568	± 0.000	2.315×10^8	$\pm 0.479 \times 10^8$	4.858	± 0.119

Table 9. Performance measures for algorithms: Intrusion dataset

Model	Runtime	± 95 %CI	Memory $\times 10^6$	± 95 %CI
skm	26442	± 239	4.594	± 0.010
skmfull	29814	± 316	4.623	± 0.012
skmhalf	27246	± 234	4.605	± 0.011
sxm	26975	± 243	4.609	± 0.011
sxmlloyds	26583	± 232	4.608	± 0.009

algorithm on our real-world datasets. Table 9, shows the relative performance of each clustering algorithm on our real-world datasets.

For the Intrusion dataset, StreamKM appears to have very high SSQ when using few clusters. StreamXM manages a more pure clustering with lower sum-of-squares than StreamKM with even a cluster per class available to it. The mean clustering of StreamXM variants is around 5, which would account for three

dominant classes. The last two clusters could be for minor classes, or also for the major classes if they are bimodal. The overall clustering purity is impressive for StreamXM in this circumstance, especially considering a total of 23 underlying classes approximated by a mean of five cluster centres.

6 Conclusions

Our proposed algorithms, StreamXM and StreamXM with Lloyds, provide statistically supported clusterings through use of the BIC. The technique is usable on data streams through using coresets to approximate the data which they run upon. We have shown that our techniques retain similar quality of clustering, especially in terms of purity, to StreamKM, while not displaying notably different behaviour in terms of runtime nor memory use. We show that when used on data stream with appearing or disappearing concepts and real-world data, we can expect similar or better quality and performance of clustering without setting a specific level of clustering for our technique to use.

References

1. Gaber, M.M., Zaslavsky, A., Krishnaswamy, S.: Data stream mining. In: Maimon, O., Rokach, L. (eds.) Data Mining and Knowledge Discovery Handbook, pp. 759–787. Springer, US (2010)
2. Miller, Z., Dickinson, B., Deitrick, W., Hu, W., Wang, A.H.: Twitter spammer detection using data stream clustering. Inf. Sci. **260**, 64–73 (2014)
3. Hanagandi, V., Dhar, A., Buescher, K.: Density-based clustering and radial basis function modeling to generate credit card fraud scores. In: Proceedings of the IEEE/IAFE 1996 Conference on Computational Intelligence for Financial Engineering, pp. 247–251. IEEE (1996)
4. Leung, K., Leckie, C.: Unsupervised anomaly detection in network intrusion detection using clusters. In: Proceedings of the Twenty-Eighth Australasian Conference on Computer Science - Volume 38. ACSC 2005, pp. 333–342. Australian Computer Society, Inc., Darlinghurst (2005)
5. Ackermann, M.R., Märtens, M., Raupach, C., Swierkot, K., Lammersen, C., Sohler, C.: Streamkm++: a clustering algorithm for data streams. J. Exp. Algorithmics **17**, 2.4:2.1–2.4:2.30 (2012)
6. Wang, C.D., Lai, J.H., Huang, D., Zheng, W.S.: Svstream: a support vector-based algorithm for clustering data streams. IEEE Trans. Knowl. Data Eng. **25**, 1410–1424 (2013)
7. Aggarwal, C.C., Han, J., Wang, J., Yu, P.S.: A framework for clustering evolving data streams. In: Proceedings of the 29th International Conference on Very Large Data Bases-Volume 29, VLDB Endowment, pp. 81–92 (2003)
8. Cao, F., Ester, M., Qian, W., Zhou, A.: Density-based clustering over an evolving data stream with noise. In: SDM. vol. 6, SIAM, pp. 326–337 (2006)
9. Zhang, T., Ramakrishnan, R., Livny, M.: Birch: an efficient data clustering method for very large databases. In: ACM SIGMOD International Conference on Management of Data, pp. 103–114. ACM Press (1996)

10. Guha, S., Meyerson, A., Mishra, N., Motwani, R., O'Callaghan, L.: Clustering data streams: theory and practice. IEEE Trans. Knowl. Data Eng. **15**, 515–528 (2003)
11. Pelleg, D., Moore, A.: X-means: Extending k-means with efficient estimation of the number of clusters. In: Proceedings of the 17th International Conference on Machine Learning, Morgan Kaufmann, pp. 727–734 (2000)
12. Lloyd, S.: Least squares quantization in pcm. IEEE Trans. Inf. Theor. **28**, 129–137 (1982)
13. Pelleg, D., Moore, A.: Accelerating exact k-means algorithms with geometric reasoning. In: Proceedings of the fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 277–281. ACM (1999)
14. Manning, C.D., Raghavan, P., Schütze, H.: Introduction to Information Retrieval. Cambridge university press, Cambridge (2008)