UNIVERSITAT POLITÈCNICA DE CATALUNYA

FACULTAT D'INFORMÀTICA DE BARCELONA

Streaming Data Clustering in MOA using the Leader Algorithm

Author:
JAIME ANDRÉS MERINO

Program for Master Thesis in:
Innovation and Research in Informatics

Master Specialization:
Data Mining and Business Intelligence

DIRECTOR: Luis Antonio Belanche Muñoz
Universitat Politècnica de Catalunya
Department of Computer Science
Research group: SOCO - Soft Computing Research Group

October 30, 2015

EXAMINING COMMITTEE

UPC - Barcelona (Spain)

October 30, 2015

PRESIDENT: Ricard Gavaldá Mestre
Universitat Politècnica de Catalunya
Department of Computer Science
Research group: LARCA - Laboratory of Relational Algorithmics, Complexity and Learnability

SECRETARY: Pedro Francisco Delicado Useros
Universitat Politècnica de Catalunya
Department of Statistics and Operations Research
Research group: ADBD - Analysis of Complex Data for Business Decisions

VOCAL: Lidia Montero Mercadé
Universitat Politècnica de Catalunya
Department of Statistics and Operations Research
Research group: MPI - Information Modelling and Processing

Abstract

Clustering is one of the most important fields in machine learning, with the goal of grouping sets of similar objects together and different from others which are placed in different groupings. Traditional unsupervised clustering tasks have been normally carried out in batch mode where data could be somehow fitted in memory and therefore several passes on the data were allowed. However, the new *Big Data* paradigm, and more precisely, its volume and velocity components, has created a new environment where data can be potentially non-finite and arrive continuously. Such *streams* of data can reach computing systems at high speeds and contain data generation processes which might be non-stationary. For clustering tasks, this means impossibility to store all data in memory and unknown number and size of clusters. Noise levels can also be high due to either data generation or transmission. All these factors make traditional clustering methods unable to cope. As a consequence, stream clustering has emerged as a field of intense research with the aim of tackling these challenges. *Clustream*, *Denstream* and *Clustree* are three of the most advanced state of the art stream clustering algorithms. They normally require two phases: first *online* micro-clustering phase, where statistics are gathered describing the incoming data; and a second *offline* macro-clustering phase, where a conventional non-stream clustering algorithm is executed using the high level statistics resulting from the *online* step. Because of their design, they either require expert-level parametrization or suffer from low runtime performance or have high sensitivity to noise or degrade considerably in high dimensional spaces because of their *offline* step. We propose a new stream clustering algorithm, the *STREAMLEADER*, based on *leader* clustering principles. It extends *cluster feature vector* abstraction capabilities and is designed to use no conventional offline clustering algorithm. This is achieved by using a novel aggressive approach based on distribution cuts, making it highly resilient to noise and allowing it to achieve very fast runtime computation while also maintaining accuracy in high dimensional settings. It works in a normalized space where it detects hyper-spherical clusters, requiring only one unique non expert user-friendly parameter. We integrate it in MOA platform (*Massive Online Analysis*), choose a sound set of seven clustering quality metrics and test it extensively against *Clustream*, *Denstream* and *Clustree* with a comprehensive set of both synthetic and also real data sets. The results are encouraging, outperforming in most of the cases the three contenders in both quality metrics and scalability.

List of Tables

1 *Clustream* main characteristics . . . 23
2 *Denstream* main characteristics . . . 24
3 *Clustree* main characteristics . . . 24
4 Conventional *Leader* clustering algorithm . . . 25
5 *CMM* measure details . . . 71
6 *Rand Statistic* quality measure details . . . 71
7 *Silhouette Coefficient* quality measure details . . . 72
8 *Homogeneity* quality measure details . . . 72
9 *Completeness* quality measure details . . . 73
10 *F1-P* quality measure details . . . 73
11 *F1-R* quality measure details . . . 74
12 *Q_AVG* quality measure details . . . 74
13 Elements to create synthetic test scenarios . . . 75
14 Synthetic quality testing: visual examples of the scenarios . . . 77
15 Synthetic quality testing: results for each scenario . . . 78
16 *Forest Covert Type* data set details . . . 89
17 *Forest Covert Type*: parametrization used for *StreamLeader*, *Clustream*, *Denstream* and *Clustree*. 91
18 *Forest Cover Type*: quality test results . . . 94
19 *Network Intrusion* data set details . . . 95
20 *Network Intrusion*: parametrization used for *StreamLeader*, *Clustream*, *Denstream* and *Clustree* 96
21 *Network Intrusion*: quality test results . . . 107

List of Figures

1 Damped or fading window model . . . 11
2 Sliding window model . . . 12
3 Landmark window model . . . 12
4 Attribute-based data stream clustering concept . . . 13
5 Instance-based data stream clustering concept . . . 14
6 MOA's logo . . . 19
7 MOA's workflow to extend framework . . . 19
8 MOA's clustering task configuration GUI . . . 21
9 MOA's clustering task visualization GUI . . . 22
10 Minkowski distances for various values of p . . . 31
11 True cluster (in black) captured by a *LeaderKernel* (in red) with its *leader* in MOA . . . 34
12 Two separate *LeaderKernels* (left) which get close enough and merge (right) . . . 35
13 Average μ distance of instances contained in a *LeaderKernel* (in red) to its *leader* is $\frac{LSD}{N}$ . . . 37
14 *LeaderKernel* (in red) not contracting in C1 (reason $\frac{LSD_1}{N_1}$ within $\frac{D_MAX}{2} \pm 20\%$) . . . 38
15 *LeaderKernel* (in red) contracts in C0 to the detected mass (reason $\frac{LSD_0}{N_0} < 80\% \frac{D_MAX}{2}$) . . . 38
16 Temporal relevance of *LeaderKernel* as Gaussian $\mu + \sigma$ of timestamps of its instances (1) . . . 41
17 Temporal relevance of *LeaderKernel* as Gaussian $\mu + \sigma$ of timestamps of its instances (2) . . . 41
18 Temporal relevance of *LeaderKernel* as Gaussian $\mu + \sigma$ of timestamps of its instances (3) . . . 42
19 *LeaderKernels* are considered if their temporal relevance falls within *horizon* . . . 42
20 Percentile cut idea: attacking noise in the tail of a distribution of *LeaderKernels* . . . 44
21 Distribution of *LeaderKernels* according to number of instances. Percentile cuts (1) . . . 45
22 Distribution of *LeaderKernels* according to number of instances. Percentile cuts (2) . . . 45
23 Manually generated distribution of *LeaderKernels*. Percentile cuts (3) . . . 46
24 Distribution of *LeaderKernels* with 0% noise. Percentile cuts (4) . . . 46
25 Logarithmic cut idea: attacking noise in the elbow of a distribution of *LeaderKernels* . . . 48
26 Logarithmic cuts on distribution of *LeaderKernels* with $D_MAX = 0.1$, with 10% noise . . . 49
27 Logarithmic cuts on distribution of *LeaderKernels* with $D_MAX = 0.2$, 10% noise . . . 50
28 Logarithmic cuts on linear slope distribution of *LeaderKernels* with $D_MAX = 0.1$, 10% noise 51

29 Intersecting *LeaderKernels* with radius D_MAX (left) merge into one bigger *LeaderKernel* (right) . . . 52
30 One (non overlapping) *LeaderKernel* with radius D_MAX (left) expands its radius (right) . . 53
31 Three non overlapping *LeaderKernels* of radius D_MAX (left) expand radius and overlap (right) (1) . . . 55
32 Three overlapping and expanded *LeaderKernels* (left) merge into a bigger one (right) (2) . . . 55
33 Flow chart of the *StreamLeader* algorithm . . . 64
34 Cluster maximum radius 0.5 (left), two of radius $\frac{0.5}{2} = 0.25$ (middle) and one 70% $\frac{0.5}{2} = 0.176$ 65
35 Clustering quality drops when a true cluster is covered by several smaller sized *LeaderKernels* 66
36 Clustering quality drops when several true clusters are captured by a single *LeaderKernels* with too large D_MAX . . . 66
37 Two *LeaderKernels* mapping true clusters (top) merge if true clusters get close enough (bottom) 80
38 micro-clusters (green) and clustering (blue) suffer distortions in high d in *Clustream*, *Clustree* using default parametrization . . . 81
39 micro-clusters (green) and clustering (blue) suffer distortions in high d in *Clustream*, *Clustree* using optimal parametrization . . . 81
40 micro-clusters (green) and clustering (blue) suffer distortions with heavy noise in *Clustream*, *Clustree* . . . 82
41 *StreamLeader* delivering clustering (in red) with default & optimal parametrization in $2d$ space 83
42 *Clustream* clustering with default & optimal parametrization (micro-clusters green, clustering red & blue) . . . 83
43 *Clustree* clustering with default & optimal parametrization. Also *StreamLeader* with optimal 84
44 Synthetic results: average *CMM* and *Silhouette Coef*, default vs optimal parametrization . . . 85
45 Synthetic results: average *F1-P* and *F1-R*, default vs optimal parametrization . . . 85
46 Synthetic results: average *Homogeneity* and *Completeness*, default vs optimal parametrization 86
47 Synthetic results: average *Rand Statistic* and overall Q_AVG, default vs optimal parametrization . . . 86
48 Synthetic results: average overall quality Q_AVG per noise levels and dimensionality . . . 88
49 Synthetic results: overall quality performance for all scenarios . . . 88
50 *Forest Cover Type*: visualizing the stream using four different sets of attributes . . . 90
51 *Forest Cover Type*: true clusters or ground-truth calculated *on-the-fly* by MOA . . . 90
52 *Forest Cover Type*: clustering by *StreamLeader* (red) and *Clustream* (blue) . . . 91
53 Forest Cover Type: *CMM* and *Silhouette Coef* default vs optimal parametrization . . . 92
54 *Forest Cover Type*: *F1-P* and *F1-R* on default vs optimal parametrization . . . 92
55 *Forest Cover Type*: *Homogeneity* and *Completeness* on default vs optimal parametrization . . 93
56 *Forest Cover Type*: *Rand Statistic* and Q_AVG on default vs optimal parametrization . . . 93
57 *Forest Cover Type*: overall quality performance . . . 94
58 *Network Intrusion*: stream visualization using two attributes sets (left & middle) and MOA's true cluster (right) . . . 96
59 *Network Intrusion*: clustering at two instants by *StreamLeader* (red) and *Clustream* (blue) . . 97
60 *Network Intrusion*: clustering by *StreamLeader* (red), *Clustream* (blue) and *Denstream* (green) 97
61 *Network Intrusion*: Connection attack and reaction of *StreamLeader* (red) and *Clustream* (blue) 98
62 *Network Intrusion*: *CMM* on default vs optimal parametrization . . . 99
63 *Network Intrusion*: *Silhouette Coefficient* on default vs optimal parametrization . . . 99
64 *Network Intrusion*: *F1-P* on default vs optimal parametrization . . . 100
65 *Network Intrusion*: *F1-R* on default vs optimal parametrization . . . 100
66 *Network Intrusion*: *Homogeneity* on default vs optimal parametrization . . . 101
67 *Network Intrusion*: *Completeness* on default vs optimal parametrization . . . 101
68 *Network Intrusion*: *Rand Statistic* on default vs optimal parametrization . . . 102
69 *Network Intrusion*: overall performance on default vs optimal parametrization . . . 102
70 *Network Intrusion*: overall performance of *StreamLeader*, *Clustream*, *Clustree* and *Denstream* 103
71 *Network Intrusion*: overall performance of *StreamLeader* and *Clustream* . . . 104
72 *Network Intrusion*: overall performance of *Denstream* and *Clustree* . . . 104
73 *Network Intrusion*: *CMM* and *Silhouette Coef* for *StreamLeader*, default vs optimal parametrization . . . 105
74 *Network Intrusion*: *F1-P* and *F1-R* for *StreamLeader*, default vs optimal parametrization . . 105

75 *Network Intrusion*: *Homogeneity* and *Completeness*, default vs optimal parametrization . . . 106
76 *Network Intrusion*: *Rand Statistic* and Q_AVG for *StreamLeader*, default vs optimal parametrization . . . 106
77 *Network Intrusion*: using *StreamLeader* as an attack warning system . . . 107
78 Scalability: number of clusters VS dimensionality, (default parametrization) . . . 108
79 Scalability & sensibility: number of clusters VS parametrization (default vs optimal) in $20d$. 109
80 Scalability & sensibility: number of clusters VS parametrization (default vs optimal) in $20d$, reduced scale . . . 110
81 Scalability & sensibility: number of clusters VS parametrization (default & optimal) in $20d$, reduced scale (2) . . . 110
82 Scalability: number of clusters VS number of instances, in $20d$, default parametrization . . . 111

Contents

1 Part 1 - Problem Contextualization **7**
1.1 Goals . . . 7
1.2 Conventional machine learning VS Big Data streaming paradigms . . . 7
1.3 Introduction to stream clustering and state of the art . . . 9
1.3.1 Notation . . . 9
1.3.2 Constraints . . . 10
1.3.3 Time Window models . . . 11
1.3.4 Attribute-based VS instance-based models . . . 13
1.3.5 Numeric domain: first *Online* abstraction phase . . . 14
1.3.6 Numeric domain: second *Offline* clustering phase . . . 16
1.3.7 Other domains: binary, categorical, text and graph stream clustering . . . 17
1.4 Technology platforms used . . . 19
1.4.1 MOA (*Massive Online Analysis*) . . . 19
1.4.2 JAVA . . . 22
1.4.3 R . . . 22
1.5 Main competitors in MOA . . . 23
1.5.1 *Clustream*, *Denstream (with DBScan)* and *Clustree* . . . 23

2 Part 2 - *StreamLeader* **25**
2.1 Conventional *Leader* clustering algorithm . . . 25
2.2 Strenghts and weaknesses of *Clustream*, *Denstream* and *Clustree* . . . 27
2.3 Design strategy . . . 28
2.4 Cluster Feature Vectors: *LeaderKernels* . . . 29
2.4.1 Framework and encapsulation . . . 29
2.4.2 Properties . . . 30
2.4.3 Proximity measure: distance function in normalized space . . . 31
2.4.4 Hyper-spherical clustering: *LeaderKernels*'s *leader* . . . 33
2.4.5 D_MAX: area of influence of a *leader* . . . 33
2.4.6 Creation (instance-based) . . . 33
2.4.7 Incremental insertion of instances to *LeaderKernels* . . . 34
2.4.8 Additive merging of two *LeaderKernels* . . . 35
2.4.9 Set artificial expansion . . . 36
2.4.10 Radius: contraction capabilities . . . 36
2.4.11 Temporal relevance . . . 39
2.4.12 Is same *LeaderKernel* . . . 43
2.4.13 Inclusion probability . . . 43
2.5 Special operations in *Offline* phase . . . 44
2.5.1 Noise treatment 1: Percentile Cut . . . 44
2.5.2 Noise treatment 2: Logarithmic Cut . . . 48
2.5.3 Expansion of intersecting *LeaderKernels* with radius D_MAX . . . 52
2.5.4 Expansion of *LeaderKernels* with radius D_MAX . . . 53
2.5.5 Expansion of intersecting *LeaderKernels* with radius artificially expanded . . . 54
2.6 Pseudocode . . . 57
2.6.1 Proximity measure . . . 57
2.6.2 *LeaderKernel* . . . 57
2.6.3 *StreamLeader* . . . 64

3 Part 3 - Testing **70**
3.1 Computing resources used . . . 70
3.2 Quality metrics . . . 70
3.3 Quality tests - Synthetic data . . . 75
3.4 Quality tests - Real Data . . . 89
3.4.1 Data Set 1: Forest Cover Type . . . 89
3.4.2 Data Set 2: Network Intrusion . . . 95
3.5 Scalability and sensitivity tests . . . 108

4 Part 4 - Conclusions and future work **112**
4.1 Conclusions . . . 112
4.2 Future work . . . 113

5 Part 5 - Appendix **114**
5.1 Stream clustering terminology . . . 114
5.2 References . . . 117

1 Part 1 - Problem Contextualization

1.1 Goals

The aim of this master thesis can be briefly outlined in three simple points:

- Develop a new stream clustering algorithm based on the Leader concept.[1]
- Integrate the new algorithm in MOA[2] (*Massive Online Analysis*) streaming platform.
- Make it a viable alternative to existing stream clustering algorithms[3].

Because streaming is a recent paradigm in the computing field and only very scarce information was at hand at the time of choosing this master thesis, first step was to do research on the status of stream technology. Only by doing that we could first know the environment and then consider what options were available in order to develop a competitive stream clustering algorithm. This entire first section is therefore dedicated to understanding the need of streaming and then the underlying concepts that we will need to master. Finally, we will analyze in detail the main competitors we will compete against in MOA. Only then we will be able to achieve our goals.

1.2 Conventional machine learning VS Big Data streaming paradigms

The world is being rapidly digitalized, and therefore large-scale data acquisition is becoming a reality. The sheer volume of data created, its heterogeneity and its velocity has created a new environment were conventional computational techniques are no longer sufficient to store and process all that information. *Big Data*, a new world describing this new paradigm, has emerged in both social and scientific communities and it is revolutionizing the world as we know it. In the field of machine learning, streaming is a new field of research were the velocity component has the main focus and has attracted a lot of attention recently.

Why is it gaining momentum?

Traditional machine learning approaches tackled the problems of prediction, classification or frequent pattern mining in an environment where it was assumed that the amount of data being generated was a finite unknown stationary distribution. That allows data to be stored physically and therefore batch mode analysis and several passes on the data was possible.

We focus particularly on Clustering, which is an important unsupervised classification technique. There is plenty of literature covering this field, like [XW08] or [ELL+10]. The goal of clustering is to group a set of n objects in k classes, homogeneous and distinct among them, which helps us uncovering structures in data that were not previously known. How homogeneous they are is normally defined by a *proximity* measure between all pairs of objects.

Conventional clustering approaches can be categorized as follows:

Direct Partitioning or Partition Representatives: this approach tries to break up the dataset into groups, attempting to minimize the distance between points labeled to be in a cluster and a point designated as the center of that cluster. *K-means* or *k-medoids* are known examples where k groups are normally known *a priori* and an element is selected as representative for its group, creating Gaussian clusters. They have linear cost and produce local optimal partitions.

Density-based: clusters are areas of higher density, i.e *DBSCAN* algorithm.

Probabilistic: such methods assume that data is originated from probability distributions. A well-known example is *EM-algorithm*. It is a parametric method that assumes that data originates from an unknown probability distribution, which then models with a mixture of distributions, number and coefficients unknown. Each component of the mixture is then identified with a cluster. *Cobweb* would be another example.

[1]Unsupervised clustering method, as outlined in [Har75]. New version of the algorithm was developed in PFC *Algoritmos de Clustering basados en el concepto de Leader*, from Jerónimo Hernández González, Facultat D'Informàtica de Barcelona, Universitat Politècnica de Catalunya, 2009.

[2]MOA, (`http://moa.cms.waikato.ac.nz/`) is an open source framework for data stream mining from the University of Waikato in New Zealand. Project leaders are Albert Bifet, Geoff Holmes and Bernhard Pfahringer and contributors Ricard Gavaldá, Richard Kirkby or Philipp Kranen among others.

[3]That is, compete in MOA against state of the art algorithms in the field.

Hierarchical: bottom-up/Top-down using dendrograms and aggregation criterion, like single linkage, complete, average linkage and Ward. They have quadratic cost and produce sub-optimal partitions (nested classes).

Sequential clustering: they take profit of hierarchical clustering, where number of classes is calculated together with corresponding centroids. Then a conventional clustering algorithm, like *k-means*, is used taking as seeds centroids previously calculated.

Algorithmic: like *Greedy/Hill-Climbing*, swapping elements between clusters.

Spectral: linear or non-linear methods use the spectrum to perform dimensionality reduction before any clustering technique is applied. In that way, factors can be used and points can then be embedded in the space, taking the structural component of the data where noise can be reduced. Interpretation is more difficult since factors are used instead of the original variables.

These techniques are powerful, have sound theory backing them and have been well established during the last few decades. They have been normally performed in batch mode since data volumes could be fitted in memory. When that was not the case, bigger data sets could be somehow handled requiring special treatment like chunking the data for parallel processing and using local clustering with final combination of results, or similar techniques.

But big sets combined with high speed in data arrival will overwhelm all the above mentioned approaches if enough data and speed is reached. We therefore need new approaches if we still want to acquire knowledge from the data we are going to receive.

Do we have such scenarios were *data streams*, potentially infinite, arrive at high speeds, need immediate processing or at the very least can not be stored due to their volume?

Below a few examples:

- NASA live satellite data.
- airplane real-time flight monitoring systems.
- network intrusion detection.
- forest fire real-time monitoring systems.
- stock market analysis.

In the examples above, effective real-time stream clustering could deliver the following:

- fully automatic sky-scanning and real-time clustering of detected stars.
- cluster airplane systems according to their functioning: *normal/abnormal*.
- clustering or detection network connections:*normal/abnormal* (attacks).
- cluster forest areas according to fire ignition metrics, like *normal/dangerous/fire ignition*.
- real-time clustering of stock shares according to high-yield metrics.

To sum it up, the world is becoming fully digitalized, everything is being measured and all that data is coming from all sort of sensors and devices, in large volumes and at very high speeds. All aspects in life, ranging from science to society and economy are impacted by this new digital era, and the goal is clear: take well-informed decisions based on the knowledge extracted from the processing of the incoming data. The need to develop advanced techniques which are able to handle that fast processing is therefore created, and this master thesis is focused on this direction, the creation of a new stream clustering algorithm.

1.3 Introduction to stream clustering and state of the art

Here, we will outline the main aspects and approaches within the streaming environment. At the same time, we will include state of the art work in the research community following each of those approaches.

1.3.1 Notation

Definition 1 (Data Stream). *A Data Stream* [Agg07], [GG07], [Gam07] *S is a massive sequence of instances* [4]*, x^1, x^2, ..., x^N, i.e, $S = \{x^i\}_{i=1}^N$ potentially unbounded ($N \to \infty$). Each instance is described by an d-dimensional attribute vector $x^i = [x^i_j]_{j=1}^d$ which belongs to an attribute space Ω that can be continuous, binary, categorical or mixed.*

Also important concepts we will use in this work are *weight* (of an instance), *horizon*, *clustering*, *ground-truth clustering*, *online phase* and *offline phase* We take the definitions as summarized in [KKJ+11] and [BF14]:

Definition 2 (Weight). *Let t_{now} be the current time and t_x the arrival time of instance x with $t_x \leq t_{now}$. Then the weight of x is $w(x) = \beta^{-\lambda \cdot (t_{now} - t_x)}$. Parameters β and λ specify the form of the decay function for the weight according to time.*

Definition 3 (Horizon). *The horizon H for a data stream S and a threshold ξ is defined as $H = \{x \in S | w(x) \geq \xi\}$.*

Definition 4 (Clustering). *A clustering algorithm takes a set of objects $O = \{x^1, x^2, ..., x^n\}$ as input and returns a cluster set $C = \{C^1, C^2, ..., C^k, C^\emptyset\}$. $x \in C^i$ implies that x lies within the cluster boundary of C^i and $C^\emptyset$ contains all unassigned objects. Objects might fall into several clusters.*

Definition 5 (Ground-truth clustering). *For a given object set $X = \{x^1, ..., x^n\}$ a set of true classes is a set $CL = \{cl^1, ..., cl^l\}$ that partitions X with $cl^i \cap cl^j = \emptyset$, $\forall i, j \in \{1, ..., l\}, i \neq j$ and $X = \{cl^1 \cup ... \cup cl^l\}$. The partitions cl^i are called classes and $cl(x)$ is the class of x. In the presence of noise given by object set $X^{noise} = \{x'^1, ..., x'^m\}$, where $X^+ = \{X \cup X^{noise}\}$, the set of true classes is the set $CL^+ = \{CL \cup cl^{noise}\}$, and $cl(x'^1), ..., cl(x'^m) = cl^{noise}$ is the noise class.*

We define the ground-truth clustering as the set $CL_o = \{cl^1_o, ..., cl^l_o\}$ with cluster cl^i_o, $\forall i \in \{1, ..., l\}$ as the smallest cluster that contains all objects from cl^i within its boundary: $\forall x \in cl^i, x \in cl^i_o$.

A ground-truth cluster cl^i_o from the ground-truth clustering CL_o might contain points from other ground-truth cluster cl^j_o as well, i.e. $cl^i_o \supseteq cl^j_o$, as two ground-truth classes can overlap, not being necessarily disjoint as objects from cl^i_o may fall into the boundaries of cl^j_o.

Definition 6 (Online Phase). *For a given data stream S formed by set of instances x^1, x^2, ..., x^N, i.e, $S = \{x^i\}_{i=1}^N$, potentially unbounded ($N \to \infty$), online phase is the summarization of the instances in stream S in real time (i.e. in a single pass over the data) by a set of k' micro-clusters $M = \{M^1, M^2, , ..., M^{k'}\}$ where M^i with $i = \{1, 2, .., k'\}$ represents a micro-cluster in a way such the center, weight and possibly additional statistics can be computed.*

Definition 7 (Offline Phase). *For a data stream S and a set of micro-clusters M, use the k' micro-clusters in M as pseudo-objects to produce a set C of $k \ll k'$ final clusters using clustering defined in Definition 4.*

[4] Also called in the literature *data objects* or *observations*. In this thesis, we will adhere to the term *instance* as it is also used in the *MOA* platform for the implementation of stream clustering algorithms.

1.3.2 Constraints

As stated before, *Big Data* streaming environment poses special challenges where following constraints apply:

- Data volumes can be high or unbounded.
- Data arrives continuously (in the form of a single *Instance* or *Observation*).
- Order of *Instance* arrival can not be controlled.
- Dimensionality of the *Instances* is not limited.
(Please note that even traditional machine learning or Multivariate Analysis techniques could bring dimensionality down, by using for instance feature selection, feature extraction, factorial coordinates PCA, CA, MCA, MDS and so on, they require considerable computation which will most likely not be available anymore in high volume/high velocity incoming streaming data).
- Only one pass will be therefore allowed on the data.
- Data is discarded after being processed the first time.
(Some relaxation exists on the matter where some of the data can be stored for a period of time with a forgetting mechanism to discard it later).
- Data might be dynamic in nature.
(That means, the generation process and probability distribution might change over time).
- Computations need to be fast and scale linearly.
(This is because the velocity of incoming data, number and size of clusters, and number of *Instances* might vary).

Successful stream clustering algorithms must therefore fulfill as many of the following requirements as possible:

- Provide timely results.
- Fast adaptation to change in underlying data distributions, also known as *Concept Drift*.
(This includes creation, evolution and disappearance of clusters. The current strategy for most data stream clustering algorithms to tackle non-stationary distributions is through the use of *Time Windows*. Also, high levels of automation should be achieved in order for the algorithm to adapt automatically to changes).
- Scalability in terms of number of instances arriving.
(Model should not grow with the number of *Instances* processed).
- Data Heterogeneity
(*Instances* being numerical but also categorical, ordinal, nominal values.
But also complete structures like DNA sequences, XML trees will need to be processed in the future).
- Rapid detection and handling of outliers and/or noise.

1.3.3 Time Window models

In order to detect potential change in a data distribution generation and keep memory and computation time down, most recent data in the stream is normally analyzed. This is necessary to assign more importance to newer instances of the stream compared to older ones. If that was not the case, change would not be detected. Clustering in a stream environment can also vary depending on the moment it is determined as well as on the *time window* or *horizon* over which they are measured. That is, clustering for the last 5 seconds might yield an entirely different clustering result as one determined for the last 5 months.

The problematic described above can be solved by adopting *time window models*. Three approaches are used in data streams where different time spans are taken when it comes to consider parts of the stream. [ZS02]: *Damped Window Models*, *Landmark Window Models* and *Sliding Window Models*.

Damped Window Models: also called *Time Fading* models, where more recent instances have higher importance than older ones. Fading is usually implemented by assigning weights to the instances so that most recent data will have higher weights. This can be implemented by using *decay* functions that scale-down the weight of instances according to the elapsed time since it appeared in the stream. An example of such function would be $f(t) = 2^{-\lambda t}$ where λ indicates the rate of decay and t the current time. Higher values of λ render lower importance to data belonging to the past. These models with decay factors are used normally in density-based stream clustering algorithms as density coefficients as we saw in the case of *D-Stream* [CT07]. In Figure 1 we see a representation of a damped window model where each instance (in blue) is allocated with a changing weight (in red), which decreases with time with a decay function.

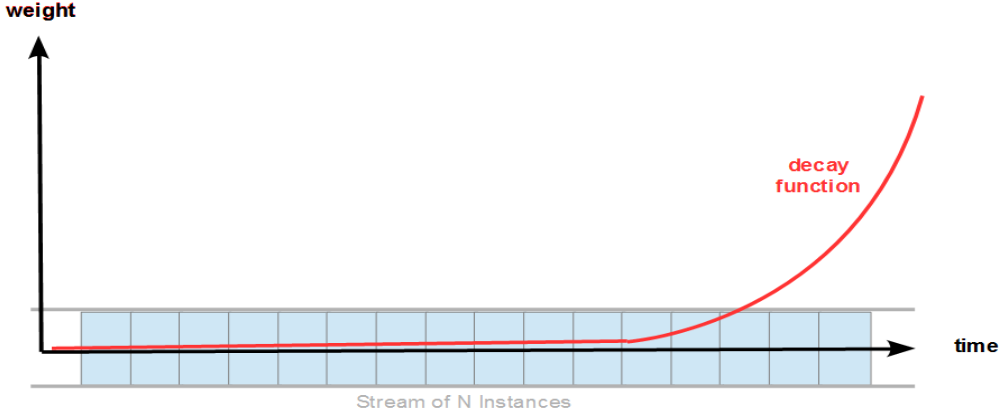


Figure 1: Damped or fading window model

Sliding Window Models: this model assumes than recent data is more relevant than older data by using *FIFO* queue concept (*First In First Out*). Accordingly, a *window* or *horizon* is maintained by keeping the most recent instances falling within the window of the data stream and discarding older ones. With the instances contained in the window, three tasks can be carried out: detect change using sub-windows, obtain statistics from the instances or update the models (only if data has changed or in any case). The size of the window is a normally a user-defined parameter. A small window size would select last few instances of the stream and therefore the algorithms would react quickly to *concept drift*, reflecting then accurately the new distribution. On the other hand, a large window size would select a larger set of instances which increase accuracy in stable periods. Finding therefore an ideal window size is a challenge, a trade-off between stability and change of non stationary distributions. To tackle this problem, a new technique named *Adaptive Window ADWIN/ADWIN2* in [BG06] has been developed, which set up change detectors in data distribution and automatically adapt window size accordingly. Experiments show very positive results which improve overall accuracy.

In Figure 2 a sliding window model is applied to a data stream with a *horizon H* of fixed length. From current time stamp t_i, the instances falling within the length of *horizon* (in blue) are taken into account for computations and the rest of the stream is discarded (in white). As two time units pass by (we move into t_{i+2}), two new instances arrive which now fall within *horizon* and the two located in the tail of the queue are dropped.

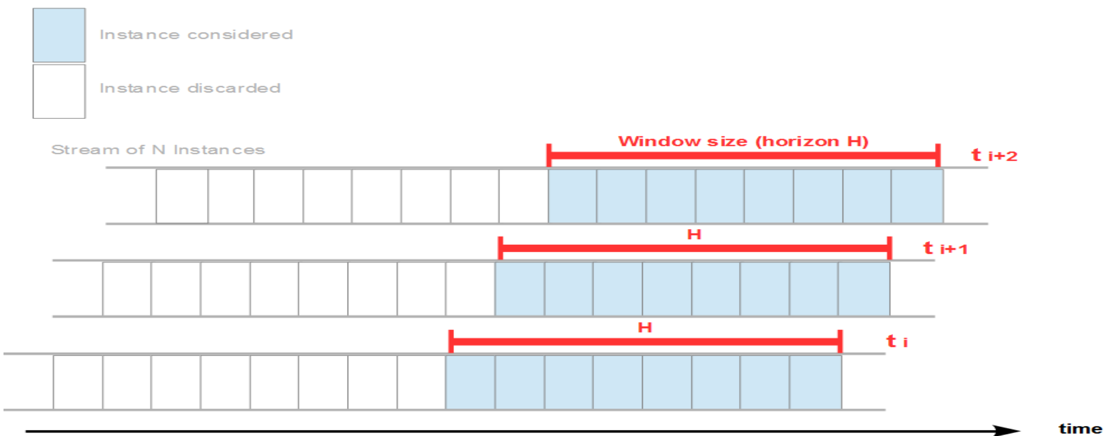


Figure 2: Sliding window model

Landmark Window Models: with this technique, portions of the data stream are separated by *landmarks*, which are relevant objets defined by time (i.e. daily, weekly) or number of instances observed as velocity factor. Therefore, non overlapping sections of the stream are taken and instances of the stream arriving after a landmark are summarized until a new landmark appears. When that happens, summarizations from the former one is discarded to free up memory. A similar approach is used in *Clustream* [AHWY03] [5] with the use of a pyramidal time frame structure where *CFs* are stored at certain snapshots or landmarks following a pyramidal pattern defined by user parametrization. Finer granularity is maintained according to recency and allowing eventually to perform clustering over different time horizons.

Such model is displayed in Figure 3, where a landmark of size 16 time units applies. Starting at time t_i, all instances are considered (in blue) until a new landmark occurs (at time t_{i+16}). Then, instances with a timestamp older than t_{i+16} are discarded and instances occurring after t_{i+16} are considered for computation until new landmark at t_{i+32} appears where the same process is repeated. At current time t (in black), instances are considered starting from t_{i+32} until future landmark appears at t_{i+48}.

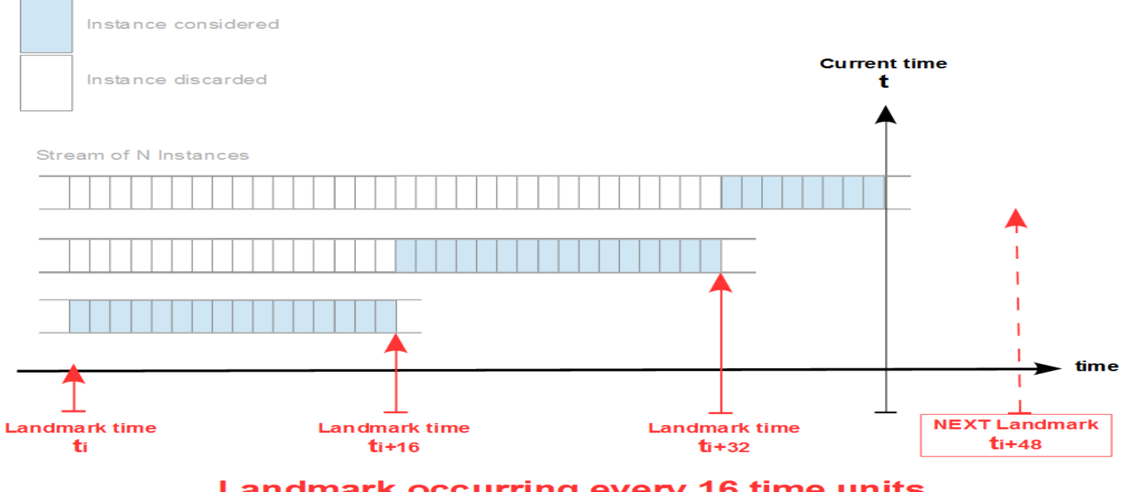


Figure 3: Landmark window model

[5] Clustream also uses decay-based statistics for its *CFs*.

1.3.4 Attribute-based VS instance-based models

Stream clustering has 2 types of applications: *Attribute-based Clustering* [6] and *Instance-based Clustering* [7].

The objective of *Attributed-based Clustering* is to find groups of attributes that behave *similarly* through time (i.e sensors). Figure 4 represents the concept of an attribute-based model.

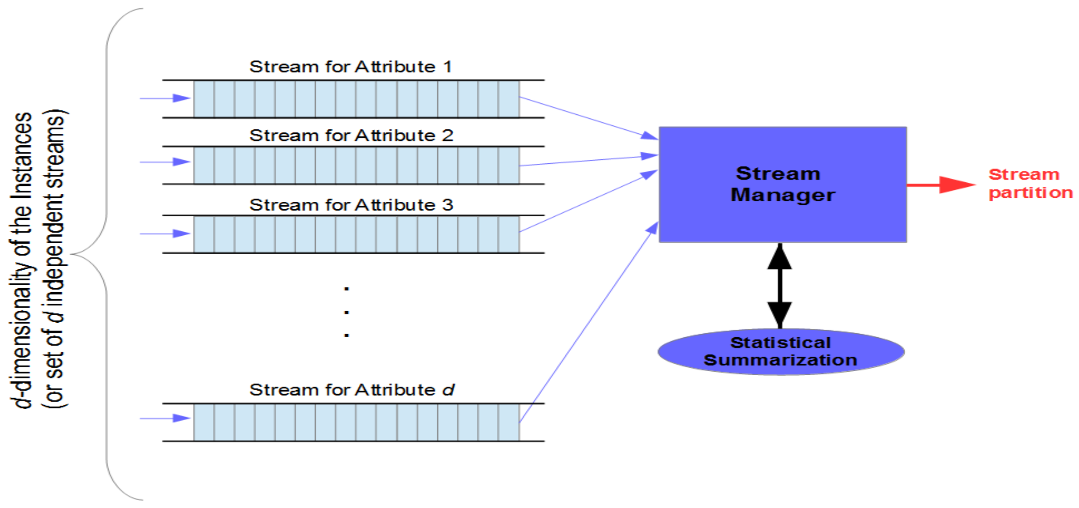


Figure 4: Attribute-based data stream clustering concept

This type of clustering had limited research but some work is being done in this area where normal clustering would be done by transposing the data matrix, which is now not possible since the stream might be non-finite. An example would be *ODAC* (*Online Divisive-Agglomerative Clustering*) [RGP06], which produces hierarchical clustering of time series data streams. Since each attribute is now a stream on its own, such clustering approaches can benefit from parallelization, like *DGClust* [GRL11], which has a central and local sites. Local sites monitor their own streams, sending them to the central site, where a global state is formed as combination of local states (grid cells) of each sensor. Finally, final clustering is performed at the central site using only the most frequent global states.

Instance-based clustering, on the other hand, clusters each instance in the stream as a whole entity containing its own attributes in the *d*-dimensional space.

It requires two separate steps, a *Data Abstraction phase* (also known as *Online step* [8]) where streaming data is summarized at a high level and a *Clustering phase* (also known as *Offline step* [9]) where final clustering is provided based on the components delivered in the *Online phase*. Most the the research is done in this area and most of the stream clustering algorithms follow this approach, like *Clustream* [AHWY03], *Denstream* [CEQZ06], *Clustree* [KABS11], *BIRCH* [ZRL97], *D-Stream* [CT07], *StreamKM++* [AMR$^+$12] among others.

We can see the conceptual representation of the model in Figure 5, where a stream feeds the abstraction module (online phase) and its output is then redirected into the clustering part (offline phase). Final clustering model is then returned.

[6] Also found in the literature as *Variable-based Clustering*.
[7] Also found in the literature as *Object-based Clustering*.
[8] In this thesis, we will refer to this step as the *Online step*.
[9] In this thesis, we will refer to this step as the *Offline step*.

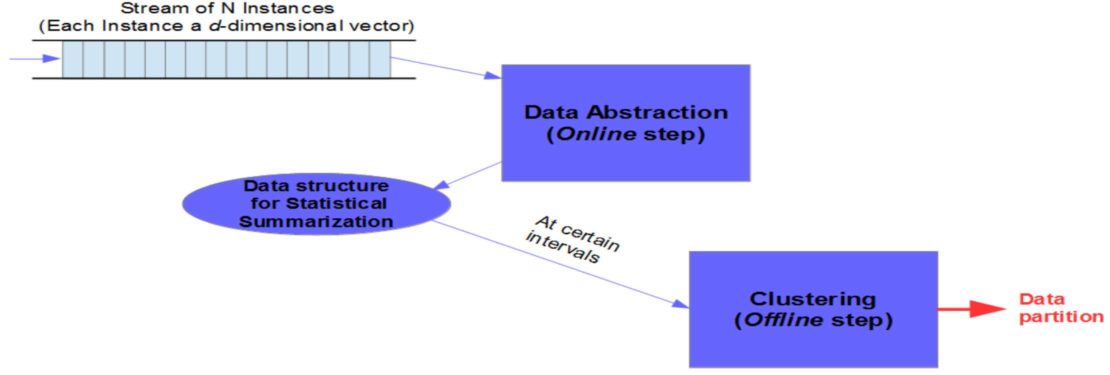


Figure 5: Instance-based data stream clustering concept

1.3.5 Numeric domain: first *Online* abstraction phase

We will focus now on stream clustering algorithms designed for the *numerical* domain, that is, in the data stream x^1, x^2, ..., x^N, i.e, $S = \{x^i\}_{i=1}^N$ potentially unbounded ($N \to \infty$), each instance being described by an d-dimensional attribute vector $x^i = [x^i_j]_{j=1}^d$, the attribute space Ω is continuous. [10]

The *Online* step is in charge of summarizing the data stream using special structures, which do not need to store the instances themselves. This is done because of memory limitations and the linear scalability that the algorithms need to have. They follow the approach of incrementally summarizing the stream. The structures most commonly used to support this phase are *Feature Vectors*, *Coresets* and *Coreset Trees*, *Prototype Arrays* and *Grids*, which are described below.

Feature Vectors: named *CF* after *Cluster Feature Vectors*, used for first time in *BIRCH* [ZRL97], it is a powerful and simple idea to capture statistics in data streams.

It does so by keeping three main components:
N: number of elements which are summarized in a *CF*.
LS: d-dimensional vector containing the Linear Sum of the objects which are summarized in a *CF*.
SS: d-dimensional vector containing the Squared Sum of the objects which are summarized in a *CF*.

Summarization is done incrementally by adding incoming instances to the corresponding vectors that already encapsulate the stream and also allowing additivity or merging of *CFs*. This sort of data abstraction is used as basis in *Clustream* [AHWY03], *Denstream* [CEQZ06], *Clustree* [KABS11] among others. In *HPStream* [AHW$^+$04], *CFs* are also used in projected clustering techniques where large number of features are available. This situation presents a special challenge because of sparsity in high dimensional spaces, where distance-based method might not be optimal. To find better clustering, a subset of relevant dimensions are determined and stored for each cluster, so that distances to those are defined only by using the set of relevant dimensions. In this sense, projected clustering could also be treated as a preprocessing step for stream processing.

[10] We will review other domains, namely binary, categorical, textual and graph-based, in next chapters.

Coresets and *Coreset Trees*:
for a given:
$|P|$: set of N instances.
d: dimensionality.
m: coreset size.
$S = \{q_1, q_2, ..., q_i\}$, where $i \leq m$: *coreset* sample points

a *coreset* is a small weighted set S, such that for any set of k-cluster centers, the (weighted) clustering cost of the *coreset* is an approximation for the clustering cost of the original set P with a smaller relative error. Seeding procedures to obtain small *coresets* from a stream were used in *k-means++* algorithm [AV07].

The main advantage is that algorithms can be applied on the much smaller *coreset* to compute an approximated solution for the original set P more efficiently. For large m (number of *coreset* points) distances need to be calculated from each point to its nearest neighbor in S, which might be too slow.

StreamKM++ algorithm [AMR+12] presented stream encapsulation by using a *Coreset Tree*, which is a more efficient structure created to speed up the computation of the *coreset* in the form of a binary, balanced tree, which is hierarchical divisible from the set P of points and has exactly m leaf nodes having each a q_i representative point that belongs to the *coreset* S. Points would only be stored at leaf level since inner nodes are defined as union of its children nodes. These nodes would contain a set of points E_i, a prototype for them q_i, number of objects in the node N_i, and a metric SSE_i indicating the Sum of Square Distances to the prototype q_i in the set of objects E_i. The advantage is that it enables the computation of subsequent sample points by taking only certain points from a subset of P into account that is significantly smaller than N. If the *Coreset Tree* is balanced, then the tree depth is $O(\log k)$, and we need time $O(dN \log m)$. These sort of algorithms contain the so-called *merge-and-reduce* steps. *Reduce* would be done via the *coreset*, and *merge* would be carried out in another structure, in this case a *bucket set*. *Buckets* store m *coreset* representatives each until completely filled in, and when several buckets are complete, a new *Reduce* step would be performed with the points contained in each *bucket*.

Grids: summarization is done in density grids, which are segmentations of the d-dimensional feature space like used in *D-Stream* [CT07], *cell trees* in [PL07] or *DGClust* [GRL11]. *D-Stream* [CT07] uses *agglomerative* clustering strategy to group grid cells. Number of points in each cell will define the density of that cell so a density coefficient is associated to each incoming instance at time t which decreases over time with an exponential decay factor. Those instances are assigned to a density grid cell. The overall density of a grid cell comes as the sum of densities associated to the instances assigned to the cell. Eventually, if the cell does not receive instances, is marked as potential outlier, and periodically removed from the list of valid cells. Strongly correlated dense cells are finally assigned together to form clusters. A parameter defines if two grids are strongly correlated. A drawback for these methods is that the number of grids is exponentially dependent on the dimensionality d so it is required to remove cells that are empty. To keep memory needs down, some other not empty cells that contain few instances will also need to be removed which might degrade the cluster quality.

Prototype Arrays: instead of the data stream as such, it can be summarized as an array of representatives, or *Prototype Array* [DH01], where each of them can be the representative of certain parts of the stream. In *Stream* [GMMO00], the stream is chunked into pieces and each is summarized by a set of representatives using a version of *K-Medoids* in [KR90].

When summarization is achieved (by using the structures outlined above), the problem remains that newer data should have more importance than older data. A general technique used to achieve this is the use of *Window Models*, which are discussed in next sections.

An additional problem would be the *Outlier Treatment* in the stream environment [B02]. An outlier is defined as an observation falling outside the overall pattern of a distribution and can occurr due to transmission problems, data collections or similar. However, in a stream, outliers could be the first instances coming from a change in underlying data generation distributions. Therefore special attention should be paid in order to treat them carefully. In the literature we can find algorithms which handle in different ways: *Clustree* [KABS11] benefits the strategy of achieving finer granularity by managing great numbers of *CFs* in the abstraction step. It does so in order to feed more precise description of the data to the *Offline*

clustering step which will handle the clusterings better with such finer granularity. Another approach would be density-based, like the one used in *Denstream* [CEQZ06] with the use of *CFs* and differentiation between *o-micro-cluster* (outlier) and *p-micro-clusters* (part of a potential cluster). A *p-micro-clusters* becomes *o-micro-cluster* if its weight falls below a certain threshold ($w < \beta\mu$) which is provided by two different parameters, β and μ. In the same way, an *o-micro-clusters* can evolve into *p-micro-clusters* which will not be considered an outlier anymore.

1.3.6 Numeric domain: second *Offline* clustering phase

In this step, final output of the stream clustering algorithm is produced. It normally involves the application of a conventional [11] clustering algorithm to find the clusters based on the statistics gathered on the online phase. In this way, the size of the data to cluster is of smaller size than the data stream being originally analyzed.

Input parameters for the algorithms vary greatly in number and complexity, being *time window size* or *horizon* needed when time window models are used, which is normally the case.

By looking at the shape of clusters delivered, we can differentiate two types of approaches: *convex stream clustering* and *arbitrary-shaped clustering.*

Convex stream clustering techniques are based normally on *k-center* clustering, being *k-means* [Maq67], following partition representative approach, one of the most widely used algorithms which can be executed on the summarizations provided, i.e *feature vectors.* Centroids for each *CF* can then be taken either as a single element to be clustered [KABS11] or influenced by weighted factors depending on the number of elements contained in the corresponding *CF* [AHWY03]. Improved versions of *k-means* also can be applied which use randomized seeding techniques to select better initial conditions and that translates in better speed and accuracy [AV06], [BF07]. The simplicity and speed of *k-means* contains also a weakness, which is the use of *Euclidean* distance as metric which tends to form hyper-spherical clusters.

Arbitrary-shaped clustering are also used. Some research has been done using non-linear kernels [JZC06] [12] where segmentation of the stream is done by using kernel novelty detection and subsequent projection into a lower dimensional space takes place. Graph-based stream clustering algorithms also exist, like *RepStream* [LL09], using representative cluster points for incremental processing where graph structures are used to model the description of changes and patterns observed in the stream. This allows a definition of the cluster boundary which is not necessarily circular like the one produced by radius based measures. Fractal clustering [BC00] could also be possible where points are placed in the cluster for which the change in the fractal dimension after adding the point is the least. Density-based approach like *DBSCAN* [EKS$^+$96] is used in *Denstream* [CEQZ06] [13] and *D-Stream* [CT07] [14]. It uses the concept of *density-connectivity* to determine final clustering. It needs additional parametrization ϵ, β, μ and when working with provided *CFs*, it takes their corresponding weights w_i and representatives c_i. It first defines *directly density-reachable* as a situation where two *CFs* have both weight w above $\beta\mu$ and $dist(c_i, c_j) \leq 2\epsilon$, where *dist* is the Euclidean distance between centers c_i and c_j and ϵ is the neighborhood considered as relevant. Then it defines whether CF_p, CF_q are also *density-reachable* when, between their representatives c_p and c_q there is a chain of *CFs* $c_{p_1}, ..., c_{p_n}$, $c_{p_1} = c_q$, $c_{p_n} = c_p$ such that c_{p_i+1} is *directly density-reachable* from c_{p_i}. Finally, it defines that CF_p and CF_q are *density-connected* when there is a CF_m with representative c_m such both c_p and c_q are *density-reachable* from c_m. In this way, *DBSCAN* can provide final clustering that is not restricted to hyper-spheres based on distances [15]) That is, because it connects regions of high density, it has the advantage that it can deliver arbitrary-shaped clusters or shapes from mixed distributions that can not be found with techniques like *k-means.*

Both *k-means* and *DBSCAN* seem to be used very frequently in the streaming literature [16]. They offer advantages and disadvantages. One of the problems we envisage already is that, in a streaming environment,

[11] Conventional in the sense of traditional batch execution where elements fit in memory and several passes are allowed on the data.

[12] Kernel-based method for clustering high dimensional streaming data that is non-linearly separable in the input space.

[13] based on *CFs.*

[14] based on *grid cells.*

[15] In most of the cases found in literature, *Euclidean* distance is widely used as a metric.

[16] Furthermore, they are the ones used in MOA for offline clustering.

underlying data generation distributions is unknown and changing. Therefore knowing in advance the number of clusters k as input for *k-means* might not be possible, being that a major drawback. In the same way, in order for *DBSCAN* to deliver arbitrary-shaped clusters, it needs to pay the price of adjusting properly a set of input parameters which define important thresholds for weights and neighborhoods. These have a great impact in the clustering delivered, therefore this might be a challenge in a streaming environment where user expert supervision might not be available (as stated in former chapters, stream clustering algorithms should aim at high levels of automation to eliminate interaction from humans as much as possible).

1.3.7 Other domains: binary, categorical, text and graph stream clustering

The stream clustering algorithms presented up to now were designed for the *numerical* domain. Other domains are of course possible. Formally, in the data stream x^1, x^2, ..., x^N, i.e, $S = \{x^i\}_{i=1}^N$ potentially unbounded ($N \rightarrow \infty$), each instance being described by an d-dimensional attribute vector $x^i = [x_j^i]_{j=1}^d$, the attribute space Ω can be, to name some of the most active in the research community, *binary*, *categorical*, *text* and *graph* or mixed. They present their specific challenges.

Binary streams can be special cases of either quantitative streams, with possible values $\{0, 1\}$, or discrete streams, with only two possible values, so any numerical algorithm could be used. However, such data tends to be sparse, so improvements in the computation of distances have been studied for such situations [Ord03].

Work for clustering *categorical data* streams was done in *StreamCluCD* [HXD+04] where incoming instances are assigned to clusters according to similarity functions. The key in this process is to maintain frequency counts of the attribute values for each cluster that are above certain threshold, while keeping memory requirements down. This is solved with the use of *approximate frequency counts* on data streams [MM02] so that only value frequencies that are *large* are maintained. When the number of possible discrete values of each attribute increases so much so that they can not be tracked in a space-limited scenario, this is referred as *massive-domain scenario*. Formally [Agg09], in the data stream x^1, x^2, ..., x^N, i.e, $S = \{x^i\}_{i=1}^N$ potentially unbounded ($N \rightarrow \infty$), each instance being described by an d-dimensional attribute vector $x^i = [x_j^i]_{j=1}^d$, with attribute space Ω categorical, the attributes value x_k^i is drawn from the unordered domain set $J_k = \{v_1^k, ..., v_{M^k}^k\}$. M^k denotes the domain size of the kth attribute and can be very large, potentially in the order of millions or billions. It is then particularly challenging to maintain intermediate statistics about such large number of attribute values. *CSketch* [Agg09] proposes a framework for these situations where frequency statistics are managed with techniques like the *count-min sketch* [CM05], which is a *sketch-based* approach to keep track of attribute-based statistics in the streaming data. It uses independent hashing functions. Same hash functions are used for each table and each map to uniformly random integers with a specific range. Sketch tables are therefore maintained for each cluster, which contain 0's at initialization point and will contain the values of the incoming records. When a new instance comes, similarity with the clusters needs to be calculated. This can be implemented by using *dot-product* between the instance and the representative of each cluster (i.e centroid). Dot-product is calculated using the d-dimensional instance with the frequency statistics stored in each cluster. Assignment to a cluster occurs when this function retrieves the largest value among all clusters. Finally, frequency counts of the cluster's representative are updated with the instance's attributes using the sketch table. There might be also collisions among the hash-cells, therefore overestimation might occur, still, results are accurate enough.

There also exists research with *text* streams. Scenarios could range from clustering webs retrieved from crawling processes to document organization. [AY06] proposes a framework to cluster massive text streams where outliers or emerging trends can be discovered. In order to create summarized representation of the clusters, it works with so-called *condensed droplets*, analogous to the summarization provided by *CFs* feature vectors in the numerical domain. They represent the word distributions within the cluster. Time-sensitive weights are assigned to instances [17] in the form of decay or fading functions, adding temporal relevance to the cluster is assigned to. Clusters fade eventually when no instances are assigned to them, favoring that the greater the number of instances in them, the harder it is for them to fade. New instances can be assigned to existing clusters by checking its similarity with them. For this purpose, the *cosine similarity* metric is used. If inserted in a cluster, its statistics are updated to reflect the insertion and temporal decay

[17] Instances in this domain could be entire text documents.

statistics. A different approach to text stream clustering is presented in [HCL+07] with the use of so-called *bursty features*, designed to highlight temporal important features in text streams, and therefore detect new emerging topics (different from clustering the underlying data stream). They are based on the concept that semantics of a new topic is marked by the appearance of a few distinctive words in the stream and that they dynamically represent documents better over time. These features are identified by using Kleinberg's 2-state finite automation model [Kle02]. Then, time-varying weights are assigned to the features by the level of *burstiness*. Identification and weighting of the bursty features are dependent on occurrence frequency. Finally, standard *k-means* is used to provide final clustering based on the new representations of the stream.

Clustering massive *graph* streams tries to bring together sound graph theory with stream processing. Graphs are very interesting structures because they are very flexible and capture information very effectively, and seem natural candidates to capture, for instance, data coming from social or information networks, which are producing large amounts of data every single day. While there are methods for clustering graphs [RMJ07], [FTT03], they are designed for static data and are not applicable with graph streams. Also, massive graphs would present serious challenges for these techniques due to the large number of nodes and edges that need to be tracked simultaneously, which translates into storage and computational problems. Graph streaming is therefore a topic of intense research. Depending on the type of incoming instance in the stream and the task to accomplish, we can distinguish two sort of situations: *node clustering* and *object clustering*. In *node clustering* situations, each incoming instance is a node or an edge of a graph (that is, we are clustering one single graph and not a stream of different graphs) and the task is to carry out is to determine general dynamic community detection. On the other hand, *object clustering* tackles situations where incoming instances are graphs drawn from a base domain, containing nodes and edges, and these graphs are clustered together based on their structural similarity.

Regarding *node clustering*, there is work related to outlier detection [AZY11], where detection focuses on structural abnormalities or different from "typical behavior" of the underlying network. This shows in the form of unusual connectivity structure among different nodes that are rarely linked together. Unusual connectivity structures can only be defined using historical connectivity structures, therefore the need to maintain structural statistics. The network is thus dynamically partitioned using structural *reservoir sampling approach* and maintains structural summaries of the underlying graph. Conceptually, node partitions represent dense regions in the graph, so rare edges crossing those dense regions are exposed as outliers. The algorithm does not handle updates in the graph (i.e deletions), only additions to the graph. While this work focuses on outlier graph detection and detects node clustering in an intermediate phase, [EKW12] provides specific dense node clustering in graph streams by refining the structural *reservoir sampling*. The algorithm handles time-evolving graphs where updates are given in the form of stream of vertex or edge additions and deletions in an incremental manner, which can be easily parallelized.

Object clustering arises from the applications where information is transmitted as individual graphs. This could be the case in movie databases where actors and films are nodes and edges are the relationships among them. In these applications, small graphs are created or received, having the assumption that the graphs contain only a small fraction of the underlying nodes and edges. This sparsity property is typical for massive domains. It gives raise to the problem of handling, representing and storing the fast incoming graphs. *GMicro* algorithm [AY10] proposes *hash-compressed micro-clusters* from graph streams. It combines the idea of sketch-based compression with micro-cluster summarization. This is very useful in cases like massive domain on the edges, where number of distinct edges is too large for the micro-cluster to maintain statistics explicitly. Graph micro-clusters contain, among others, a sketch-table of all the frequency-weighted graphs which are added in the micro-cluster. Intermediate computations, like distance estimations, can then be performed as sketch-based estimates while maintaining effectiveness. Additional work to discover significant (frequently co-occurring) and dense (node sets that are also densely connected) subgraphs in the incoming stream shows in [ALY+10]. A probabilistic *min-hash* approach is used to tackle sparsity, summarize the stream and efficiently capture co-occurrence behavior efficiently or patterns.

1.4 Technology platforms used

1.4.1 MOA (*Massive Online Analysis*)

As stated before, the requirements for this master thesis is the creation of a new stream clustering algorithm, its integration in MOA platform (*Massive Online Analysis*) and benchmark analysis against state of the art competitors.

Figure 6: MOA's logo

We accessed the available documentation[18] ([BHK+11], [BK12], [BK+12], [Str+13]) and check MOA's capabilities so that we make sure that it has the proper features we need for the project:

- MOA is a specialized platform for data stream mining developed by the University of Waikato, New Zealand. [19]. It is well-known and accepted in the research community and includes a collection of machine learning algorithms, ranging from classification, regression, clustering, outlier detection, *concept drift* detection and recommender systems. Project leaders are Albert Bifet, Geoff Holmes and Bernhard Pfahringer and contributors Ricard Gavaldá, Richard Kirkby or Philipp Kranen among others.

- MOA has portable architecture. Related to the WEKA project, it can scale to more demanding problems. This is achieved by using JAVA as programming language.

- MOA is open source, follows modular design able to accommodate new algorithms and metrics. This is done via interfaces:

Figure 7 shows the internal modules and workflow used in MOA.

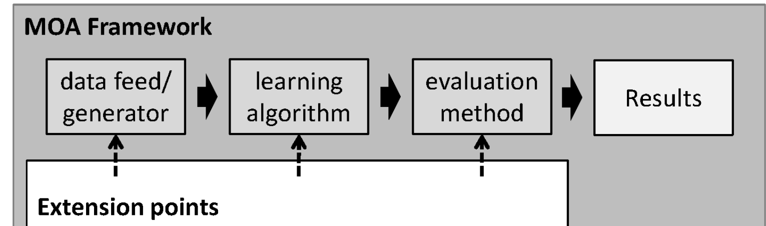


Figure 7: MOA's workflow to extend framework

We will therefore *plug-in* our logic in two places in the framework:

- *data feed/generator*: here we will feed real data to be turned into a stream so that we can benchmark our algorithm against the others.
- *learning algorithm*: we will integrate our new algorithm in this section.

[18] `http://moa.cms.waikato.ac.nz/documentation/`
[19] (`http://moa.cms.waikato.ac.nz/`)

- MOA provides an application program interface API. It is then possible to use methods of MOA inside JAVA code. We will use these in order to implement our algorithm and integrate within MOA.

To add a new stream clustering algorithm, we need to implement the *Clusterer.java* interface with the following three main methods:

void resetLearningImpl(): a method for initializing a clusterer learner.
void trainOnInstanceImpl(Instance): a method to train a new instance.
Clustering getClusteringResult(): a method to obtain the current clustering result for evaluation or visualization.

- MOA generates synthetic streaming data[20]: synthetic stream data generation is achieved through RBF (Radial Basis Function) data generators [21], which are normalized in the [0,1] range.

- MOA streaming of real data provided by user[22]: real data can be fed to MOA and turned into a stream by providing a data file and adding headers that specify how to stream the different fields and/or potential normalization of the data.

- MOA has basic stream visualization capabilities[23] by choosing specific dimensionality. Still, visualizing multi-dimensional streaming data is challenging and a field of research on its own.

- MOA has the ability to run several competing algorithms in parallel[24].

- MOA can gather, visualize and export clustering quality statistics, based on a set of metrics available for clustering evaluation[25].

- MOA allows to perform tasks via command line interface:

this is important for our scalability and sensitivity tests. As an example, the command

EvaluateClustering -l clustream.Clustream -s (RandomRBFGeneratorEvents -K 2 -k 0 -R 0.025 -n -E 50000 -a 5) -i 500000 -d dumpClustering.csv

evaluates the runtime performance of the *Clustream* algorithmm with a synthetic feed random RBF generator, creating 500000 data instances for the stream, of 2 clusters, with no variation in number of clusters, cluster radius 0.025 in normalized space, no creation/deletion/merge/split cluster event, which would otherwise occur every 50000 instances[26]. The results would be dumped into the dumpClustering.csv file (for further analysis).

[20] See figures 8 for configuration, section 3.
[21] See Appendix on *Streaming Terminology* for further details.
[22] See figures 8 for configuration, section 3.
[23] See figure 9 for visualization, section 8.
[24] See figures 8 for configuration, section 3.
[25] See figure 8 for export, figure 9 for visualization, section 9.
[26] There are other values that are set by default, like no variation in cluster's radius, noise level 10 percent, etc.

- Finally, we will launch the MOA[27] application from the command line. First indicate where the software is located:

$cd\ c:\backslash MOA_2014_11$
$set\ MOADIR = C:\backslash MOA_2014_11$

then set the $CLASSPATH$ variable so that JAVA can find the *.jar* files:

$set\ CLASSPATH = .;\%MOADIR\%\backslash moa.jar;\%MOADIR\%\backslash sizeofag.jar;\%CLASSPATH\%$[28]

and finally launch the application to start up the main GUI:

$java\ -javaagent:sizeofag.jar\ moa.gui.GUI$

MOA's task configuration GUI is shown in Figure 8.

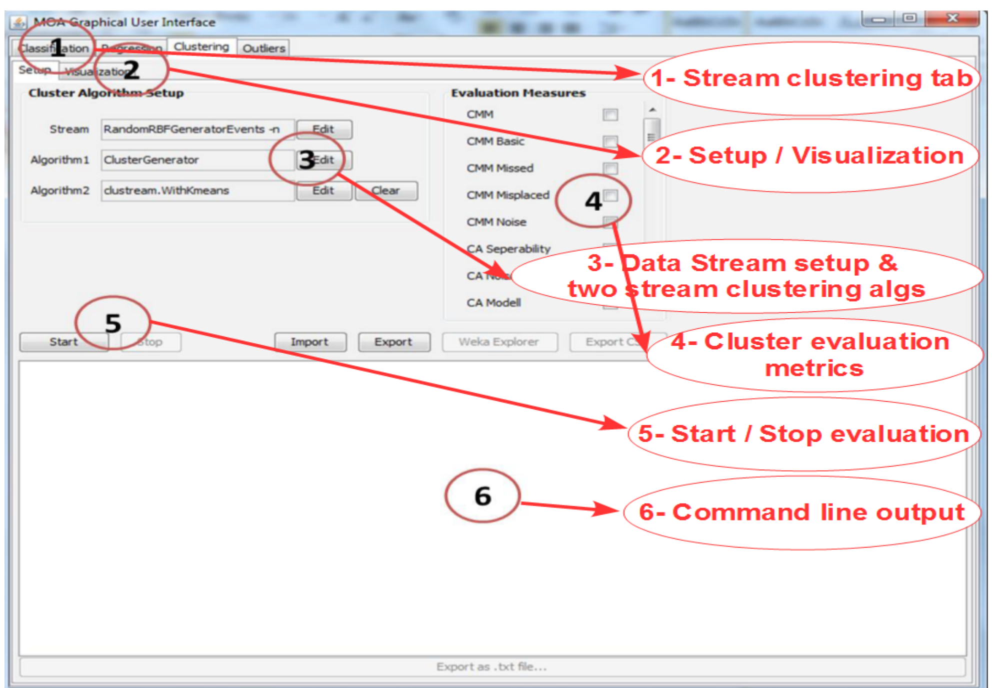


Figure 8: MOA's clustering task configuration GUI

[27] *MOA Release 2014.11* (`http://moa.cms.waikato.ac.nz/downloads/`) was used since it was the latest official release at the time of starting this thesis. At the time of writing this document, a new pre-release was available *MOA Pre-Release 2015.05*

[28] For the implementation of the new algorithm, we will use a *JAVA* mathematical library called *commons-math3-3.5.jar*. The Apache Commons Math project is a library of lightweight, self-contained mathematics and statistics components addressing the most common practical problems not immediately available in the Java programming language or commons-lang. Therefore the additional *.jar* component will be added to the *CLASSPATH* (`http://commons.apache.org/proper/commons-math/download_math.cgi`)

And the corresponding clustering task visualizatio in Figure 9:

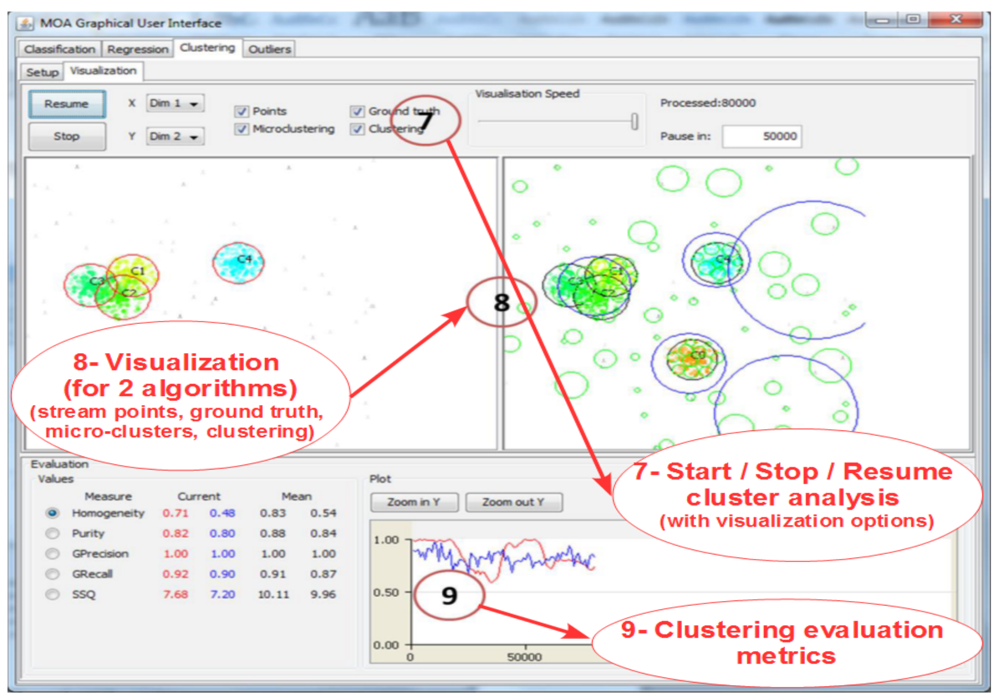


Figure 9: MOA's clustering task visualization GUI

1.4.2 JAVA

We need to integrate our algorithm in MOA, which is implemented in *JAVA*. We will use *Eclipse*:

Eclipse IDE for Java Development
Version: Kepler Release
Build id: 20130614-0229

We will compile the classes containing our new algorithm:

javac LeaderKernel.java
javac LeaderStream.java

then place the generated *.class* files (*LeaderKernel.class* and *LeaderStream.class*) under the directory created for the new logic [29],

$\%MOADIR\%\backslash moa\backslash clusterers\backslash StreamLeader$

and finally launch the application to start up the main GUI with the new algorithm showing in the application[30]:

$java\ -javaagent : sizeofag.jar\ moa.gui.GUI$

1.4.3 R

In order to carry out statistical analysis and plots needed for the design, analysis and graphs used for tests visualization of the results from MOA, *R* was chosen.

R is a free software environment for statistical computing and graphics. It compiles and runs on a wide variety of UNIX platforms, Windows and MacOS. The software version:

[29]Respecting MOA's folder structure for the location of stream clustering algorithms.
[30]See Figure 8: MOA's clustering task configuration GUI, area *3* in circle, where the new algorithm should be now available.

R version 3.0.1 (2013-05-16).

We used *R Studio* as IDE for *R*:

R Studio, an open source *IDE* for *R* development

Version 0.97.551

1.5 Main competitors in MOA

The stream clustering algorithms available in MOA[31] are are *Clustream*, *Clustree*, *Denstream (with DBScan)*, *CobWeb* [32]([Fis87]) and *StreamKM++* [33].
Cobweb crashes when being executed and *StreamKM++* does not seem to deliver the quality metrics. We will therefore choose *Clustream*, *Clustree*, *Denstream (with DBScan)* as contenders.
Information about the three algorithms was already outlined in former sections while explaining the different aspects of stream clustering environment. In next section, Table 1, Table 2, Table 3 summarize briefly their main characteristics in MOA (MOA's implementation is sometimes not identical to original published papers).

1.5.1 *Clustream*, *Denstream (with DBScan)* and *Clustree*

Algorithm in MOA	**Clustream**
Type	Stream clustering algorithm (partition representatives)
Domain	Numeric
Cluster Model	Instance-based
Data Structure	Cluster feature vectors
Time Window Model	Sliding window
MOA parameters	2 - $MaxNumKernels$ (number micro-clusters) - $KernelRadiFactor$ (multiplier for kernel radius)
Cluster Algorithm	k-means
Cluster Shape	Hyper-sphere
Brief Description	It uses extension of CFs to summarize data, called micro-clusters. CFs are assigned time stamps, so only recent are kept as active. Number of CFs and size is specified by the user. It creates a pyramidal time frame to store the CFs as snapshots (MOA implementation does not use this framework). Allows clustering analysis over different horizons based on the pyramidal time frame (in MOA time horizon is fixed). The framework has different levels of granularity, newer data stores more frequently than older data. Offline clustering needs two snapshots from pyramidal time frame (MOA clusters based on active micro-clusters). k-means is used to provide clustering based on stored CFs. Number of clusters either provided by MOA (true number) or by user.

Table 1: *Clustream* main characteristics

[31]In *MOA Release 2014.11*
[32]*Cobweb* is a hierarchical probability-based clustering algorithm. Probability-based approaches make assumptions on probability distributions on attributes. It incrementally organizes objects into a classification tree by classifying the object along the path of best matching nodes. Each node is a class (concept) and is labeled by a probabilistic concept. The tree is therefore used to predict the class of an object. A related problem is that often the tree is not height-balanced. For skewed input distributions this poses a problem in performance. In any case, it is not a stream clustering algorithm, that is the reason it was not mentioned up to now in this document. Still, it appears in MOA as an implemented algorithm, although it crashes when being executed.
[33]StreamKM in the menu.

Algorithm in MOA	**Denstream**
Type	Stream clustering algorithm (density-based)
Domain	Numeric
Cluster Model	Instance-based
Data Structure	Cluster feature vectors
Time Window Model	Damped or fading window
MOA parameters	7 - ϵ (neighborhood around micro-clusters) - β (outlier threshold) - μ (weight of micro-clusters) - $initPoints$ (initialization) - $offline$ (multiplier for ϵ) - λ (defines decay factor) - *processingSpeed* (incoming points per time unit)
Cluster Algorithm	DBSCAN
Cluster Shape	Arbitrary-shaped
Brief Description	It uses CFs or micro-cluster to summarize the data. No limit on number of micro-clusters but constrain on radius & weight. p-micro-clusters keep track of potential clusters. o-micro-clusters keep track of potential outliers. Weight time aging fading for p-micro-clusters, using decay function. p-micro-clusters below a threshold turns into o-micro-clusters. Offline clustering based on DBSCAN. It builds the concept of density reachable & density connected regions. Number of clusters non predefined.

Table 2: *Denstream* main characteristics

Algorithm in MOA	**Clustree**
Type	Stream clustering algorithm (partition representatives)
Domain	Numeric
Cluster Model	Instance-based
Data Structure	Cluster feature vectors
Time Window Model	Damped or fading window
MOA parameters	1 - $MaxHeight$ (maximum height of the tree)
Cluster Algorithm	k-means
Cluster Shape	hyper-sphere
Brief Description	It uses CFs to summarize the data. It uses fast indexing structure, R Tree, to store and maintain CFs. Temporal information is assigned to nodes and CFs which decay with time. R Tree allows computation of distances in logarithmic time, therefore insertions are logarithmic in the number of CFs maintained. Entries in node are split into 2 groups so that sum of intra-group distances is minimal. The further down in the tree, the more fine-grained the resolution of the micro-clusters. There are different descend strategies inside the tree. It implements any-time clustering, interrupting the process any time can deliver clustering. With enough time, instances descend through the tree to the most similar CF. When no such CF is found, a new is created. Buffers in nodes for instances whose descend was interrupted. They are called Hitchhikers. New instances descending can take Hitchhikers with them. Offline algorithm can be anything, k-means or DBSCAN (k-means in MOA).

Table 3: *Clustree* main characteristics

2 Part 2 - *StreamLeader*

Now that we reviewed the state of the art and chose the platforms for development, we focus on the design of the new stream clustering algorithm.

2.1 Conventional *Leader* clustering algorithm

As stated in the goals section, basic requirement for the new algorithm is that it must follow the principles of the Leader clustering algorithm in batch mode, as specified by John Hartigan in Clustering Algorithms in [Har75]. In Table 4 we can find the algorithm as it was originally presented and also adapted to structured programming:

INPUT: T maximum distance around each leader cluster
OUTPUT: L List of clusters

I: actual instance
M: total number of instances
J: actual cluster
K: number of clusters
$P(x)$: function that indicates to which cluster belongs instance x
$L(x)$: function that indicates the Leader of cluster number x
T: maximum distance d_max (as a distance metric)
(function d would be defined as *Euclidean* distance metric)

Leader algorithm (as presented by Hartigan)	Leader algorithm (adapted to structured programming)
start 1 $I := 1$ 2 $K := 1$ 3 $P(I) := K$ 4 $L(K) := I$ 5 $I++$ 6 **if** $I > M$, **stop** 7 **else if** $I \leq M$ **then** 8 $J := 1$ 9 **if** $d(I, L(J)) > T$ 10 **goto 15** 11 **elseif** $d(I, L(J)) \leq T$ **then** 12 $P(I) := J$ 13 **goto 5** 14 **endif** 15 $J++$ 16 **if** $J \leq K$ 17 **goto 9** 18 **elseif** $J > K$ **then** 19 $K++$ 20 $P(I) := K$ 21 $L(K) := I$ 22 **goto 5** 23 **endif** 24 **endif** **end**	**start** 1 $I := 1$ 2 $K := 1$ 3 $P(I) := K$ 4 $L(K) := I$ 5 **while** $I \leq M$ **do** 6 $J := 1$ 7 **while** $J \leq K$ **and** $P(I) = 0$ **do** 8 **if** $d(I, L(J)) \leq$ T **then** 9 $P(I) := J$ 10 **end if** 11 $J++$ 12 **end while** 13 **if** $P(I) = 0$ **then** 14 $K++$ 15 $P(I) := K$ 16 $L(K) := I$ 17 **end if** 18 $I++$ 19 **end while** **end**

Table 4: Conventional *Leader* clustering algorithm

As we can see, the Leader is a partition clustering algorithm, very straightforward and fast. We outline its behavior:

As input, it receives (in its original version) a distance threshold T. This is the influence of each leader in its neighborhood. In this case, the influence was defined as *Euclidean* distance.

As output, it returns a list with the resulting clusters.

At the very beginning, first instance creates a cluster on its own and represents it as the leader of that cluster.
When a cluster is created for the first time, the instance that created it remains as the leader of that cluster, forever. That is, it will be its representative and will not change, even if more instances are assigned to that cluster.

When a new instance is taken for analysis, it is checked against the clusters.
That means, it is checked against the leaders that represent each cluster, one by one until we find the first cluster (leader) which area of influence includes the instance. We notice that ordering is quite important in two ways. The order in which checking is done against the leaders could influence the assignation to one cluster or another that also fulfills the proximity criteria. On the other hand, the ordering of the instances decides which ones create clusters and leaders (first instances usually create the clusters) and which instances are assigned to those already existing clusters. Different ordering in batch processing produces different clustering.

We also notice that checking area of influence is dependent of the similarity metric that can be used. In this case, the original algorithm was designed to work with distances (parameter T). But we see that the definition of this similarity metric is quite important since it defines how clusters are formed.

When an instance is assigned to a cluster, the leader of that cluster is not altered because of the new addition.
If the instance is checked against all leaders and none of them is close enough (similar enough), then a new cluster is created with the new instance as the leader.

We can already draw some basic conclusions:

- Our new stream clustering algorithm needs to be inspired in the leader concept, specifically in how clusters are created from instances that were not assigned to any cluster and how leaders represent them.

- We already notice that the definition of a leader is quite static; they are assigned once and remain fixed (keep if physically) for the entire existence of a cluster. This will be for sure a problem in a dynamic streaming environment with *concept drift* and *instance processed only once* policy.

- A decisive factor is the ordering. As stated already, in a streaming environment we will have no control on the ordering of the incoming instances.

- We should keep its simplicity and make it as fast as possible.

- We already know we should receive a parameter specifying the influence of the leader on the surrounding area, which in this case is *Euclidean* distance. We will need to consider what proximity measure we will use.

- Last but not least, we should try as hard as possible to create high quality clustering in a streaming environment because we will compete against state of the art *Clustream*, *Denstream* and *Clustree*.

- We will name our new algorithm *StreamLeader*

2.2 Strenghts and weaknesses of *Clustream*, *Denstream* and *Clustree*

Before taking on the design of the new *StreamLeader*, we want to come up with a list of key characteristics we want the new algorithm to have. That is, make it as robust, fast and accurate as possible while eliminating potential weaknesses. In order to do that, a good strategy is to carefully analyze the competing algorithms and see what advantages and disadvantages they posses. This will be a good starting point to start designing, by detecting their strengths and weaknesses (if any).

We analyze with spacial care the original research papers for *Clustream*, *Denstream* and *Clustree* and also their *JAVA* implementation in MOA. Furthermore, we take into considerations the techniques and algorithms reviewed in the state of the art section.

We draw some interesting conclusions based on their design:

Key strengths:

- Online phase: they achieve data abstraction quite successfully through use of extensions of Cluster Feature Vectors. Other algorithms use *grid* cells, which would be more oriented to density-based solutions which is not what we aim for. Furthermore, *tree coresets*, while effective, are complicated structures to construct and maintain, which might take a toll on performance and simplicity.

- Shape: *Denstream* can produce arbitrary-shaped clustering using the offline clustering step. *Clustream* and *Clustree* build hyper-spheric Gaussian constructs (which is more restrictive).

Key weaknesses:

- Offline phase. They all share it and is always present when clustering determination is required at a specified moment. They all require standard batch clustering technique (like *k-means*, *DBSCAN*, etc) to produce final clustering based on the data abstraction provided by online step. This applies to *Clustream*, *Denstream*, *Clustree*. We tend to think that it slows down the whole process.

- Complexity in parametrization: *Denstream* would require an expert to fine-tune the set of several parameters in a changing streaming environment, which seems a considerable challenge. On the other hand, *Clustream* requires a number of micro-clusters proportional to the number of natural clusters (also called *micro-ratio*) of, at leasts, 10 to 1. Such ratio should be unknown and also changing.

- Speed issues: maintainance of high number of micro-clusters. *Clustream*, *Denstream* and *Clustree* are based on the assumption that the more micro-clusters they can (efficiently) manage, the finer the data abstraction granularity would be passed to the offline components for final clustering, and therefore the better the quality of the clustering produced. We tend to think that more micro-clusters might provide more granularity, but also more memory consumption. Also, if we could envisage an alternative in order to not use conventional algorithm for final clustering, maintaining great numbers of micro-clusters would not be needed at all.

- Noise issues: If *CF micro-clusters* map also noise in the online step, it is also natural to think that the offline clustering algorithm based on the micro-clusters might have difficulties discarding noise from data belonging to true clustering. Determined clustering might be then affected by noise. Noise management might be an issue for some of the algorithms, transparent for the user and ideally not dependent on any user parametrization.

- Multidimensionality and accuracy: The fact that micro-clusters abstract sets of data instances in the d-dimensional space and are later used as points for clustering determination (using conventional clustering), lead us to think that high dimensional spaces would yield bigger and bigger multi-dimensional micro-clusters. If that is the case, they would potentially fill more and more the space. On top, offline clustering based on over-inflated micro-clusters might tend to produce over-sized clustering, with clusters overlapping among them. It is natural to think that, if occurring, we would also expect low levels of *recall* and *accuracy* in clustering metrics.

2.3 Design strategy

With these factors in mind, we design the *StreamLeader* with the aim of keeping or enhancing the strengths and eliminating or at least diminishing the weaknesses they all seem to share.

We aim to integrate the following key pieces in the StreamLeader's design:

- Online component: *clustering feature vectors CFs.* We use the *CF* concept to create abstractions of the data, what are called in the literature micro-clusters. We will extend them with additional capabilities to confer them as much flexibility as possible. We will name our special micro-clusters *LeaderKernels*, each one representing a potential natural cluster.

- Online component: each dimension in d space with range [0,1]. Our algorithm and the *LeaderKernels* will work by scaling the d-dimensional space observations of the stream into a space where each dimension (attribute) is restricted to [0,1]. This concept fits well with MOA synthetic data generation, which generates random RBF generators[34], Gaussian hyper-spheres, constraining each artificial attribute to [0,1] values in each dimension. Furthermore, real stream data can always be converted or approximated into that space (as MOA does). An important factor is that we plan to give our generated clusters flexibility to expand/contract if necessary to better adjust to the detected data. Working in the space described above maximizes our chances of achieving best results. Lastly, even if visualization will always be very restricted with conventional techniques, scaling down the data allows visualization and comparison of the data and clusters in MOA by selecting any two dimensions.

- Online component: spherical Gaussian-shaped clusters. The *StreamLeader* will create hyper-spherical Gaussian-shaped clusters. They will try to capture the center of masses of the detected clusters, starting off initially with a maximum radius provided by the user (D_MAX parameter).

- Online component: dynamic cluster size adaptation: We will give the *LeaderKernels*, the capability to dynamically override (contract/expand) its radius to a certain extend, with the aim to capture the data structures with as much accuracy as possible. This will be different from standard merging in some situations which is possible in other algorithms.

- Offline component: no offline conventional clustering algorithm to deliver final clustering. We will not use offline standard batch clustering algorithm in batch mode (i.e *k-means*, *DBSCAN*, etc) as it is used by all other competitors. We want in this way to create a solution conceptually different to the other streaming algorithms. Also, we aim for speed.

- Offline component: cluster size adjustment. When *time window* expires, *StreamLeader* will check potential merging of *close-enough LeaderKernels* to produce better final clustering.

- Offline component: noise elimination. When *time window* or *horizon* expires, *StreamLeader* will target specific statistical noise elimination automatically. We aim at two things: to be very resilient to noise and at the same time provide an alternative to offline conventional clustering based on *CFs*.

- Parametrization: only one user-friendly parameter will be required, D_MAX. This parameter will indicate the area of influence that a *leader* from a *LeaderKernel* has in its neighborhood. Since we work in normalized d-dimensional space, it will be a distance measure[35], with a range (0...0.5]. It determines the level of clustering to be produced. Smaller D_MAX would discover small data structures and bigger D_MAX would discover bigger ones within the data stream. Still, it will not be rigid as in the conventional *Leader* algorithm, *StreamLeader* takes this parameter into consideration and tries to adapt the size of the *LeaderKernels* to the stream, adjusting to changing environments and detecting bigger or smaller structures if it makes sense. This capability should render the algorithm much needed flexibility, having in mind that we will not use conventional clustering as last step. This one-parameter approach therefore aims at simplicity and understandability by any non expert user and the algorithm will use it in a flexible way.

[34] See *Appendix* with stream clustering definitions.

[35] Further details about this important point will be discussed in next sections.

2.4 Cluster Feature Vectors: *LeaderKernels*

2.4.1 Framework and encapsulation

The data stream needs to be captured in some way without storing the single instances. We adhere to the concept of *one-pass* algorithms where instances are only treated once and then discarded. In order to do that, we need to capture statistical information that describes the data stream. We achieve that by constructing the *LeaderKernels*, as extensions of the cluster feature vectors *CFs*.

Definition 8 (LeaderKernel). *We call a LeaderKernel in a d-dimensional normalized space, for a set of instances x^1, x^2, ..., x^n, each described by an d-dimensional attribute vector $x^i = [x^i_j]^d_{j=1}$ which belongs to an attribute space Ω that is continuous, with timestamps of arriving instances T^1,T^2, ..., T^n, with distances*[36] *to leader*[37] *(or LeaderKernel's representative) of arriving instances D^1,D^2, ..., D^n as the tuple*

$(LS, SS, LST, SST, LSD, D_MAX, N,$
$creation_timestamp, is_artificially_expanded, artificially_expanded_radius)$

where

- LS: *vector of d entries.*
For each dimension, linear sum of all the instances added to the LeaderKernel. The p-th entry of LS is equal to $\sum_{j=1}^{n} x_p^j$.

- SS: *vector of d entries.*
For each dimension, squared sum of all the instances added to the LeaderKernel. The p-th entry of SS is equal to $\sum_{j=1}^{n} (x_p^j)^2$.

- LST: *linear sum of the timestamps of all instances added to the LeaderKernel, equal to* $\sum_{j=1}^{n} T^j$.

- SST: *squared sum of the timestamps of all instances added to the LeaderKernel, equal to* $\sum_{j=1}^{n} (T^j)^2$.

- LSD: *linear sum of the distances to the leader of all instances added to the LeaderKernel, equal to* $\sum_{j=1}^{n} D^j$.

- D_MAX: *influence of the LeaderKernel, taken its leader as representative, in its neighborhood in the normalized d-dimensional space, measured in terms of the distance function, with range (0...0.5].*

- N: *Number of instances added to the LeaderKernel. Also designated as weight in this document.*

- *creation_timestamp*: *timestamp that indicates when the LeaderKernel was created.*

- *is_artificially_expanded*: *boolean that indicates whether the LeaderKernel was artificially expanded farther than indicated by D_MAX.*

- *artificially_expanded_radius*: *radius of the LeaderKernel as a result of an artificial expansion process.*

We can refer to the *LeaderKernel* for a set of instances S by $\overline{LK(S)}$.

[36] Discussion on suitable proximity measures and distance function used will be explained in next sections.

[37] Discussion on *leader* calculation will be explained in next sections. As we will see later, in original Hartigan's algorithm, the *leader* of a cluster was one of the physical points being analyzed in the point set and was static. On the other hand, *StreamLeader* will store no physical instance as such. Instead it will use the summarization stored in the *LeaderKernel* to calculate the leader dynamically.

Instead of storing the individual stream instances assigned to it, only the $\overline{LK(S)}$ vector is stored as summary, which consumes much less memory. It is also accurate enough since it contains sufficient information to calculate the measurements we need. At any point in time, it is therefore possible to maintain the summary of the dominant *LeaderKernels* describing the most *relevant* data from the stream.

2.4.2 Properties

LeaderKernels have two important properties: incremental and additive maintenance.

Property 1 (*LeaderKernel* Incremental property): *we assume that*

$\overline{LK_1(S_1)} = (LS_1, SS_1, LST_1, SST_1, LSD_1, D_MAX_1, N_1,$
$creation_timestamp_1, is_artificially_expanded_1, artificially_expanded_radius_1)$

and I an instance, arriving at timestamp T_I, at distance D_I to leader of LeaderKernel represented by vector $\overline{LK_1(S_1)}$.
Then the resulting $\overline{LK_1(S_2)}$ vector that is formed by adding the instance I to $\overline{LK_1(S_1)}$ is:

$\overline{LK_1(S_2)} = \overline{LK_1(S_1)} + I = ($
$LS_1 + I$ *on each dimension d*
, $SS_1 + I^2$ *on each dimension d*
, $LST_1 + T_I$
, $SST_1 + T_I^2$
, $LSD_1 + D_I$
, D_MAX_1
, $N_1 + 1$
, $creation_timestamp_1, is_artificially_expanded_1, artificially_expanded_radius_1)$

Property 2 (*LeaderKernel* Additivity property): *we assume that*

$\overline{LK_1(S_1)} = (LS_1, SS_1, LST_1, SST_1, LSD_1, D_MAX_1, N_1,$
$creation_timestamp_1, is_artificially_expanded_1, artificially_expanded_radius_1)$

and

$\overline{LK_2(S_2)} = (LS_2, SS_2, LST_2, SST_2, LSD_2, D_MAX_2, N_2,$
$creation_timestamp_2, is_artificially_expanded_2, artificially_expanded_radius_2)$

Then the resulting $\overline{LK_1(S_3)}$ vector that is formed by merging $\overline{LK_2(S_2)}$ to $\overline{LK_1(S_1)}$ is:

$\overline{LK_1(S_3)} = \overline{LK_1(S_1)} + \overline{LK_2(S_2)} = ($
$LS_1 + LS_2$ *on each dimension d*
, $\frac{(SS_1 * N_1) + (SS_2 * N_2)}{(N_1 + N_2)}$ *on each dimension d*
, $\frac{(LST_1 * N_1) + (LST_2 * N_2)}{(N_1 + N_2)}$
, $\frac{(SST_1 * N_1) + (SST_2 * N_2)}{(N_1 + N_2)}$
, $(LSD_1 * N_1) + (LSD_2 * N_2)$
, D_MAX_1
, $N_1 + N_2$
, (if $N_1 \neq 1$, $creation_timestamp_1$, otherwise $creation_timestamp_2$)[38]
, ($is_artificially_expanded_1$ OR $is_artificially_expanded_2$)
, max($artificially_expanded_radius_1$, $artificially_expanded_radius_2$)

[38] *LeaderKernels* with highest weight will drive merging operations, keeping their creation date. In situations where a newly created *LeaderKernels* with its first instance merges with an existing one, the later keeps the creation date.

LeaderKernels summarize information about instances contained in them by using incremental properties (adding new instances to an existing LeaderKernel) or additive (merging two *LeaderKernels*). At any moment, statistical information about the instances contained in the relevant *LeaderKernels* are maintained. When a *LeaderKernel* summarizes instances that are not *relevant* anymore, then they are eliminated by means of elimination of the corresponding *LeaderKernel*. This is the case of either noise or out-of-date information. These two points will be discussed in detail in next sections.

2.4.3 Proximity measure: distance function in normalized space

Since clustering require grouping homogeneous but distinct objects among them, a key aspect for most of the clustering algorithms in general is the proximity measure between pairs of objects in the space we are working. It reflects the actual proximity between instances according to the final aim of the clustering. Weighting the attributes might be sometimes necessary.

Before deciding which proximity measure *StreamLeader* and *LeaderKernel* will use, we can review what options we have. Then we will take an *informed decision* on this important step.

Proximity measures range from distance to distance indexes, similarity and dissimilarity indexes, depending on the properties they hold. Distances are very popular in literature. When it comes to dealing with numerical data, *Minkowsky* distances in L_p *norm* can be used, where relation between norm and distance is

$$D(I^1, I^2) = ||I^1 - I^2||$$

and *L_p norm* is the distance induced by norm L_p

$$L_p norm \rightarrow D_p(I^1, I^2) = ||I^1 - I^2||_p = (\sum_{i=1}^{d} |I_i^1 - I_i^2|^p)^{1/p}$$

in a d-dimensional space
and particular cases, for instance,

$$L_1 \rightarrow D_{Manhattan}(I^1, I^2) = ||I^1 - I^2||_1 = \sum_{i=1}^{d} |I_i^1 - I_i^2|$$

$$L_2 \rightarrow D_{Euclidean}(I^1, I^2) = ||I^1 - I^2||_2 = \sqrt{\sum_{i=1}^{d} (I_i^1 - I_i^2)^2}$$

$$L_\infty \rightarrow D_{Chebyshev}(I^1, I^2) = ||I^1 - I^2||_\infty = \max_i |I_i^1 - I_i^2|$$

We can see in Figure 10 the kind of unit circles generated in the space for various values of p[39]. They show the contour of the equidistances:

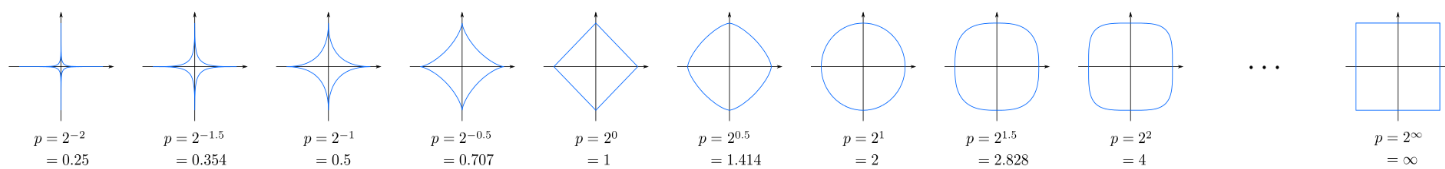


Figure 10: Minkowski distances for various values of p

Proximity measures are taken according to clustering needs. While all distances shown above are equidistant to the centers of the unit circles, L_2 $D_{Euclidean}$ results from scalar product and is a natural choice to represent distances in d-dimensional space[40]. Many stream algorithms choose L_2, *StreamKM++* [AMR+12], *Clustream* [AHWY03], *Clustree* [KABS11] *Denstream* [CEQZ06] and *BIRCH* [ZRL97], among others.

[39] Picture taken from Wikipedia: `https://en.wikipedia.org/wiki/Minkowski_distance`.
[40] It can also be more sensitive to outliers than other distance definitions.

Another important aspect to consider is MOA. It generates synthetic data by creating numerical (i.e. not textual, not heterogeneous) streams. This already constrains the sort of measure we can create since it must be based on numeric attributes. Also, the *on-the-fly* calculation of the ground-truth seems to be based on distances by creating the smallest hyper-sphere that can capture the instances belonging to the same class.

Still, we can already glimpse into the future with further possibilities in terms of creating distances for logical, categorical data or strings data for example.

So we are ready to design our distance measure. As we stated before, *StreamLeader* will work, for the time being, in a normalized d-dimensional space, where each dimension is within [0,1], with instances described by an d-dimensional attribute vector $x^i = [x^i_j]^d_{j=1}$ which belongs to an attribute space Ω continuous.

L_2 $D_{Euclidean}$, if d large enough, can render distance values greater than 1. In order to fit into a normalized space, we will constrain those values to a maximum of 1. The pseudo-code for the distance function looks as follows:

FUNCTION: $DIST_{SL}$
INPUT: I^1, I^2 stream data instances in d-dimensional space (or any object in that same d-dimensional space, like a *leader* representing a *LeaderKernel*)
OUTPUT: D distance between the two given instances I^1, I^2

1 $d \leftarrow$ {dimensionality of I^1}
2 $D \leftarrow \sqrt{\sum_{i=1}^{d}(I^1_i - I^2_i)^2}$
3 $D \leftarrow \frac{D}{D+1}$
4 **return** D

Theorem 1 ($DIST_{SL}$ metric distance): *if* L_2 $D_{Euclidean}$

$$\sqrt{\sum_{i=1}^{d}(I^1_i - I^2_i)^2}$$

is a metric distance with the following properties for S *a massive sequence of instances,* x^1, x^2, ..., x^N, *i.e,* $S = \{x^i\}^N_{i=1}$, *each instance described by an d-dimensional attribute vector* $x^i = [x^i_j]^d_{j=1}$ *which belongs to an attribute space* Ω *that is continuous*

$DIST_{SL} : S \times S \rightarrow \mathbb{R}^+$

1. $\forall x^i \in S$ $\quad DIST_{SL}(x^i, x^i) = 0$
2. $\forall x^i, \forall x^j \in S$ $\quad DIST_{SL}(x^i, x^j) = 0 \rightarrow x^i = x^j$
3. $\forall x^i, \forall x^j \in S$ $\quad DIST_{SL}(x^i, x^j) = DIST_{SL}(x^j, x^i)$
4. $\forall x^i, \forall x^j, \forall x^k \in S$ $\quad DIST_{SL}(x^i, x^k) \leq DIST_{SL}(x^i, x^j) + DIST_{SL}(x^j, x^k)$

then $\frac{D}{D+1}$ *and therefore* $DIST_{SL}$ *are also a metric distance with the same properties.*

2.4.4 Hyper-spherical clustering: ***LeaderKernels*'s *leader***

With the distance function above, the clusters created will be of hyper-spherical shape. Checking proximity of an incoming instances to the *LeaderKernel* will be done by applying the distance function between the instance and the *leader* (representing the *LeaderKernel*). In non-streaming environments, a cluster's representative could be calculated in different ways. Sometimes data points from the original set of points can be taken as representatives, like *k-medoids* in [KR87]. Such approaches are not possible in pure conceptual streaming, which adheres to the concept that *data is discarded after being processed first time*. Therefore incoming instances should not be stored, even if they act as representatives of a whole cluster.[41] *Leader* calculation is simple using the statistical summaries:

FUNCTION: *GET_LEADER*
INPUT: $\emptyset$
OUTPUT: L as statistical center of the *LeaderKernel* in d-dimensional space

1 $L \leftarrow \{\frac{LS}{N}$ on each dimension$\}$
2 **return** L

2.4.5 D_MAX: area of influence of a *leader*

StreamLeader will only require one user-defined parameter, which is the influence of the *leader* in its d-dimensional space neighborhood, with range (0..0.5] on each of the d dimensions. The influence or proximity will be measured in terms of the distance function we defined above. It determines the level of clustering to be produced. Smaller D_MAX would discover smaller data structures and bigger D_MAX would discover bigger ones within the data stream. As mentioned before, *StreamLeader* will have the flexibility to discover and adjust to clusters of larger and smaller size than the area specified by radius D_MAX. We will discuss these capabilities in next sections.

2.4.6 Creation (instance-based)

With the definitions above, we can already specify how a *LeaderKernel* can be created from one instance:

PROCEDURE: $CREATE_{INSTANCE_BASED}$
INPUT: I stream data instance, d dimensions, TS timestamp for addition of I to *LeaderKernel*, $DMAX$ radio of influence of this *LeaderKernel*, $TSTMP_creation$ timestamp for creation of *LeaderKernel*
OUTPUT: $\emptyset$

1 $N \leftarrow 1$
2 $LS \leftarrow \{I$ on each dimension$\}$
3 $SS \leftarrow \{I^2$ on each dimension$\}$
4 $D_MAX \leftarrow DMAX$
5 $LST \leftarrow TS$
6 $SST \leftarrow TS^2$
7 $creation_timestamp \leftarrow TSTMP_creation$
8 $LSD \leftarrow 0$
9 $is_artificially_expanded \leftarrow$ false
10 $artificially_expanded_radius \leftarrow 0$

[41] Relaxation on this rule can however be applied although is not the preferred approach in state or the art algorithms.

2.4.7 Incremental insertion of instances to *LeaderKernels*

As stated in Property 1, *LeaderKernels* can be maintained incrementally by absorbing incoming instances from the stream and updating their statistical summarization data:

PROCEDURE: $INSERT_INSTANCE$
INPUT: I stream data instance, TS timestamp for addition of I to *LeaderKernel*, i_DTL distance from I to *LeaderKernel*'s *leader*
OUTPUT: $\emptyset$

1 $N \leftarrow N + 1$
2 $LST \leftarrow LST + TS$
3 $SST \leftarrow SST + TS^2$
4 $LS \leftarrow \{LS + I \text{ on each dimension}\}$
5 $SS \leftarrow \{SS + I^2 \text{ on each dimension}\}$
6 $LSD \leftarrow LSD + i_DTL$

In Figure 11, the representation in MOA of a set of instances (in green), which belong to a true cluster or ground-truth cluster $C1$ (contour in black), and a *LeaderKernel* capturing the cluster (in red). The *leader* of the *LeaderKernel* is calculated as $\frac{\overline{LS_1}}{N}$ in the multi-dimensional space. D_MAX is the user-defined parameter and specifies the area of influence of the *leader*.

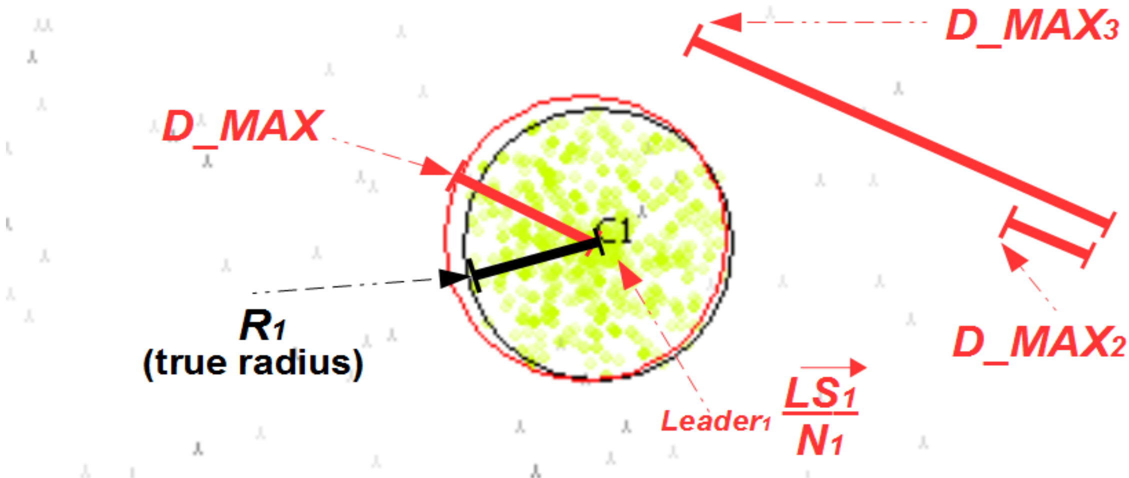


Figure 11: True cluster (in black) captured by a *LeaderKernel* (in red) with its *leader* in MOA

2.4.8 Additive merging of two *LeaderKernels*

As stated in Property 2, additive property implies that two different *LeaderKernels* can merge by absorbing one the other. Their statistical summarizations are then combined following the pseudo-code below:

PROCEDURE: *MERGE*
INPUT: lk_{add} *LeaderKernel* to be absorbed
OUTPUT: $\emptyset$

1 $radius \leftarrow$ {get radius}
2 $radius_lk \leftarrow$ {get radius from lk_{add}}
3 $creation_timestamp_lk \leftarrow$ {get creation timestamp in lk_{add}}
4 $N_lk \leftarrow$ {num instances contained in lk_{add}}
5 $LS_lk \leftarrow$ {LS in lk_{add}}
6 $SS_lk \leftarrow$ {SS in lk_{add}}
7 $LST_lk \leftarrow$ {LST contained in lk_{add}}
8 $SST_lk \leftarrow$ {SST contained in lk_{add}}
9 $LSD_lk \leftarrow$ {LSD contained in lk_{add}}
10 $is_artificially_expanded_l \leftarrow$ {check if lk_{add} was artificially expanded}
11 $creation_timestamp \leftarrow$ {if $N \neq 1$, $creation_timestamp$, otherwise $creation_timestamp_lk$ }
12 $N \leftarrow N + N_lk$
13 $LST \leftarrow \frac{(N*LST)+(N_lk*LST_lk)}{(N+N_lk)}$
14 $SST \leftarrow \frac{(N*SST)+(N_lk*SST_lk)}{(N+N_lk)}$
15 $LSD \leftarrow (N * LSD) + (N_lk * LSD_lk)$
16 $is_artificially_expanded \leftarrow is_artificially_expanded$ **or** $is_artificially_expanded_lk$
17 $artificially_expanded_radius \leftarrow$ {max between $radius$ and $radius_lk$}
18 $LS \leftarrow LS + LS_lk$
19 $SS \leftarrow \frac{(N*SS)+(N_lk*SS_lk)}{(N+N_lk)}$

In Figure 12, on the left, we observe true clusters $C0$ and $C1$ formed by corresponding instances in a data stream. *LeaderKernels* $\overline{LK_0}$, $\overline{LK_1}$ (in red contour) capture them. On the right, due to *concept drift*, $C0$, $C1$ move close enough to merge. $\overline{LK_0'}$ is the resulting *LeaderKernel* containing the combined statistics of $C0$, $C1$ and capturing the combined mass. We observe how the center of gravity of $\overline{LK_0'}$ needs to be re-calculated after the merging.

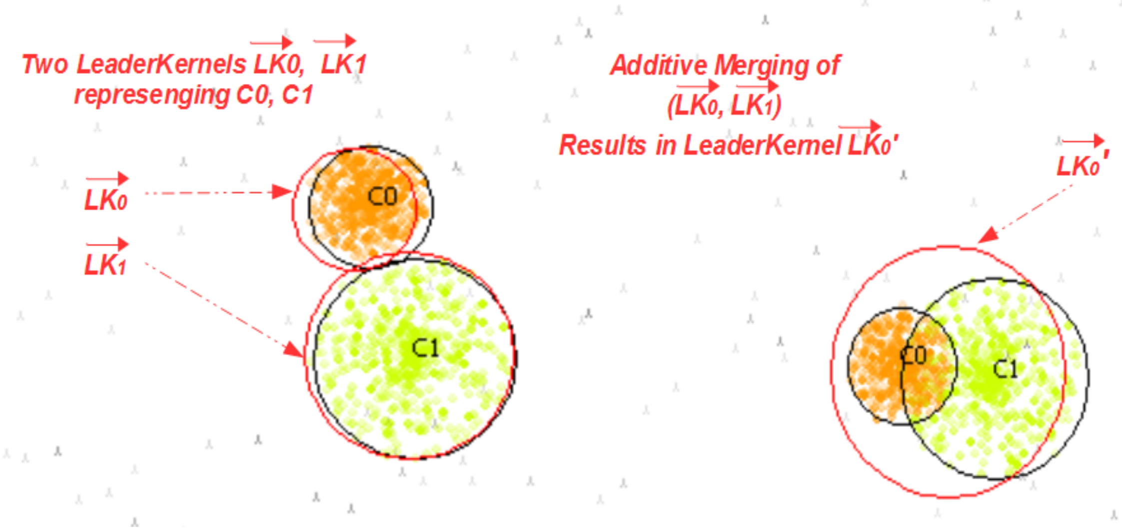


Figure 12: Two separate *LeaderKernels* (left) which get close enough and merge (right)

2.4.9 Set artificial expansion

As we will see in next sections, *LeaderKernels* can be artificially expanded. This will happen when *StreamLeader* decides that D_MAX is not large enough to capture a specific true cluster that a *LeaderKernel* is already detecting but can not capture entirely[42]. In these special cases, a larger radius than D_MAX will be given to try capturing as much instance mass[43] as possible[44].

PROCEDURE: $SET_ARTIFICIALLY_EXPANDED$
INPUT: IAE boolean indicating whether the *LeaderKernel* is artificially expanded
OUTPUT: $\emptyset$

1 $is_artificially_expanded \leftarrow IAE$

FUNCTION: $IS_ARTIFICIALLY_EXPANDED$
INPUT: $\emptyset$
OUTPUT: IAE boolean indicating whether the *LeaderKernel* is artificially expanded

1 $IAE \leftarrow is_artificially_expanded$
2 **return** IAE

PROCEDURE: $SET_ARTIFICIALLY_EXPANDED_RADIUS$
INPUT: AE_Radius artificially expanded radius to be assigned to this *LeaderKernel* (overrides D_MAX as radius)
OUTPUT: $\emptyset$

1 $artificially_expanded_radius \leftarrow AE_Radius$

2.4.10 Radius: contraction capabilities

The aim of the *LeaderKernel* is to detect mass (well-defined groups of instances) in the *d*-dimensional space, and it will try to place the *leader* in the center using the *GET_LEADER* function we already defined. There might be situations when *LeaderKernel* detects a true cluster and places itself properly in space but the area of influence specified by D_MAX could be too large to cover that natural cluster. In those cases, the delivered cluster would cover the true cluster and, either empty space (if we are lucky) or just noise or worse elements from other clusters (which would take a hit in quality metrics like *Precision*[45] and *Recall*[46])

We therefore need to handle situations occurring when true clusters are smaller than what *LeaderKernel* covers with D_MAX. That is, we allow the *LeaderKernel* to contract in radius to adjust as much as possible to the size of the natural cluster.

First we calculate the average distance of all instances added to the *LeaderKernel* to its center (we recall *LSD* contains the linear sum of the distances to the center of all instances added to the *LeaderKernel*):

FUNCTION: $GET_\mu_DISTANCE_TO_LEADER$
INPUT: $\emptyset$
OUTPUT: D average distance to the leader representing the *LeaderKernel* (of all distances of all instances to leader at the time they were added to the *LeaderKernel*)

1 $D \leftarrow \frac{LSD}{N}$
2 **return** D

Second, if the *LeaderKernel* was artificially expanded[47], then we return the artificial radius assigned to

[42] Further details on this when we explain the flow of the *StreamLeader* algorithm.
[43] The terms *weight* and *mass* are equal in this context and refer to the number of instances allocated to each *LeaderKernel*.
[44] *LeaderKernels* might be allocated with a larger radio than D_MAX in specific situations.
[45] For detail definition of the metric, refer to quality metric section.
[46] For detail definition of the metric, refer to quality metric section.
[47] Further details on artificial expansion will be explained in next sections.

it. If not, we try to adjust the radius accordingly to the summarized average distance of the mass concentrated in the *LeaderKernel*. We therefore need to be able to do that adjustment. Different densities in natural clusters can occur. If we thought of a situation where the cluster contains the instances more or less in a homogeneous way, then, the instances located close to the center of the hyper-spheres would have distances close to zero. On the other hand, instances located very close to the maximum boundary (radius R) would have distances close to R. We can then suppose that, if the cluster adjusts well to the group of instances, the average distance of all instances contained would be close to $\frac{R}{2}$, which is what $\frac{LSD}{N}$ or $GET_\mu_DISTANCE_TO_LEADER$ tries to represent. If we were to adjust D_MAX perfectly for that cluster, then $\frac{D_MAX}{2}$ would match $\frac{R}{2}$ and also $\frac{LSD}{N}$.

In Figure 13, $C0$ and $C1$ appear as true clusters formed by the corresponding instances and radius R. If two *LeaderKernels* were to capture the true clusters with accuracy, then their *leader* would be located very close to the center of the true clusters. If an instance is located in those centers, then $DIST_{SL}(instance, leader) \approx 0$. An instance located in the contour of the hyper-sphere would have $DIST_{SL}(instance, leader) \approx R_{TRUE_RADIUS}$. Lastly, an instance located at around 1/2 of the length of the true radius, would have $DIST_{SL}(instance, leader) \approx \frac{R_{TRUE_RADIUS}}{2}$. If the instances are homogeneously distributed in the cluster, we can then calculate the average distance μ of all instances to a center of a *LeaderKernel* as $\frac{LSD_0}{N_0}$ in $C0$ and $\frac{LSD_1}{N_1}$ in $C1$. This guides us in approximating the size of the true cluster. As we will see in next section, we will use this information to assign a radius to the *LeaderKernels*.

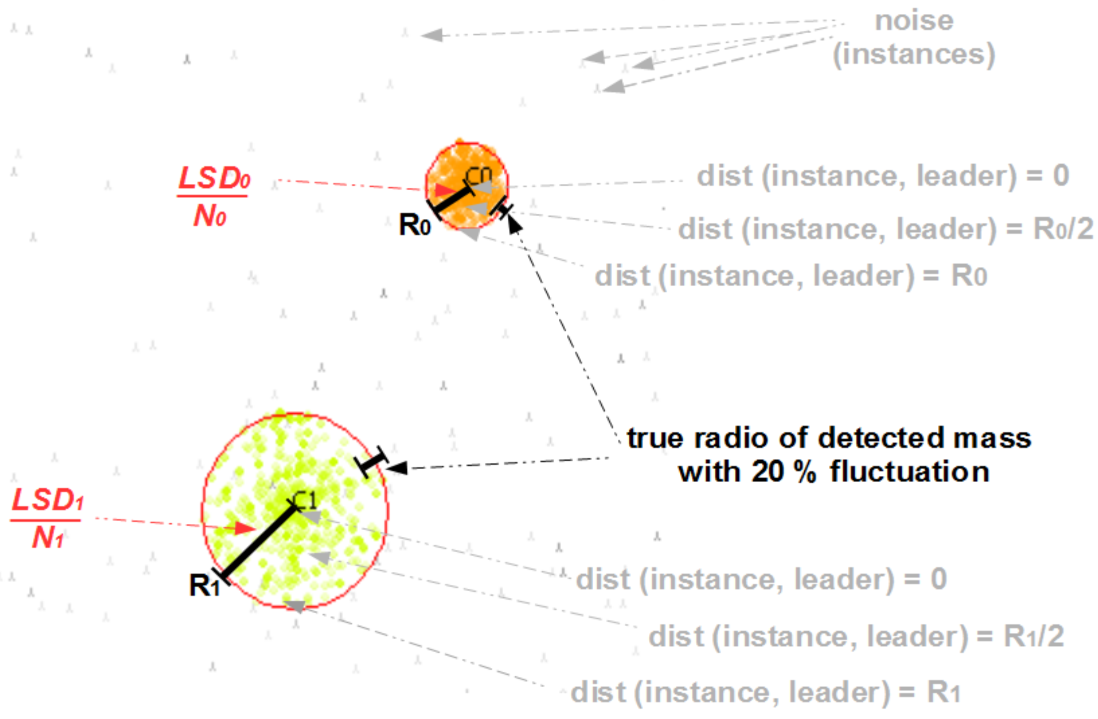


Figure 13: Average μ distance of instances contained in a *LeaderKernel* (in red) to its *leader* is $\frac{LSD}{N}$

If we had a *LeaderKernel* with D_MAX influence that captures a cluster and it is off up to 20% on the size of the detected mass (ideally detected mass would match true cluster), then we consider that D_MAX is a good radio for the *LeaderKernel* and a good estimation of the radio of the true cluster. That is, we are comfortable in regions of D_MAX between 80% and 120% of the volume of instance mass we detect. The fact that we take 20% as threshold and no other comes as an *informed decision* since we work in a normalized space, work with fast incoming and changing streaming data, work with statistical summarizations of the stream and we might have different grades of sparsity of data within the same cluster, among others. So 20% flexibility seems a reasonable value.[48]

In Figure 14 we see that *LeaderKernel* $\overline{LK_1}$ (contour in red) does not contract its predefined D_MAX

[48] Of course this threshold could change, but we prefer to carry out extensive sensibility analysis on the parametrization of the algorithm instead of the amount of flexibility we allocate to the *StreamLeader*.

radius and captures the true cluster $C1$ (contour in black). No contraction occurs because $\frac{LSD_1}{N_1}$ falls within 20% of maximum deviation of $\frac{D_MAX}{2}$, which indicates that *LeaderKernel* $\overline{LK_1}$ captures with enough accuracy the true cluster $C1$.

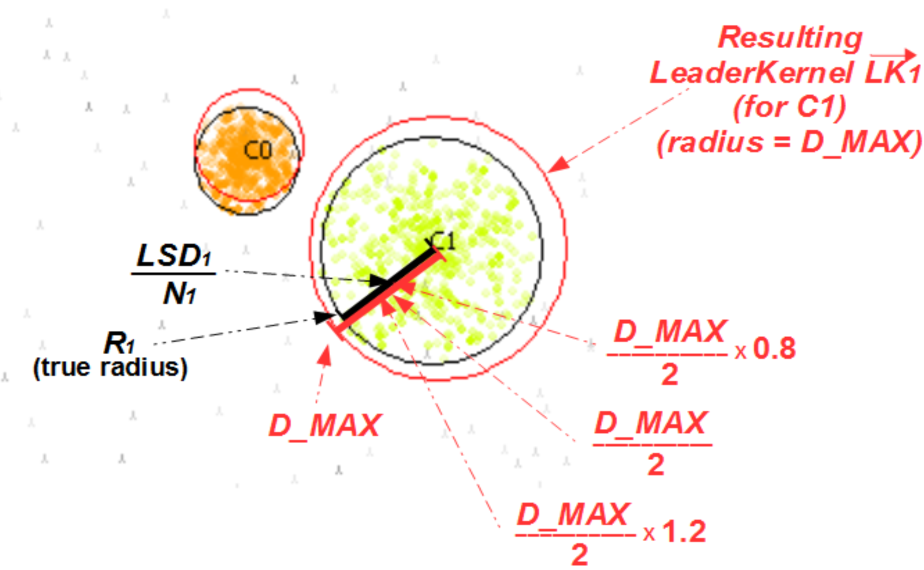


Figure 14: *LeaderKernel* (in red) not contracting in C1 (reason $\frac{LSD_1}{N_1}$ within $\frac{D_MAX}{2} \pm 20\%$)

On the other hand, Figure 15 shows a situation where *LeaderKernel* $\overline{LK_0}$ (contour in red) contracts its predefined D_MAX radius in order to capture true cluster $C0$ (contour in black) better. Contraction takes place because $\frac{LSD_0}{N_0}$ falls outside the maximum deviation of 20% allowed for $\frac{D_MAX}{2}$. This indicates that D_MAX is a too large radius to capture $C0$ and *LeaderKernel* $\overline{LK_0}$ needs to contract to a smaller radius.

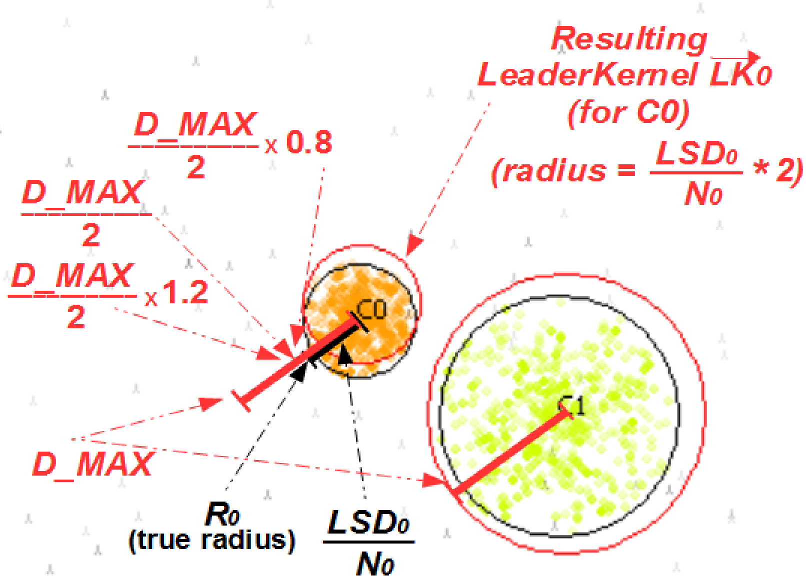


Figure 15: *LeaderKernel* (in red) contracts in C0 to the detected mass (reason $\frac{LSD_0}{N_0} < 80\% \ \frac{D_MAX}{2}$)

The logic behind the calculation of the adjustment is therefore based on the summarization of the distances maintained applied to the hyper-spherical spheres of the *LeaderKernels*. If D_MAX is off up to 20% of detected mass, then we take D_MAX as acceptable radio[49]. If it deviates more than 20% and the detected mass seems of smaller size, then we return the radio we detect using only mass ($\frac{LSD}{N} * 2$) and ignoring the guideline given by D_MAX. The pseudo-code below shows how the radius is calculated. We therefore allow *LeaderKernels* to contract in radius as follows:

FUNCTION: GET_RADIUS
INPUT: $\emptyset$
OUTPUT: R radius of the *LeaderKernel*

1 $Mu_D \leftarrow$ {get μ distance to leader}
2 $R \leftarrow 0$
3 **if not** $is_artificially_expanded$ **then**
4 **if** $Mu_D < (\frac{D_MAX}{2} * 0.8)$ **then** $R \leftarrow Mu_D * 2$
5 **else** $R \leftarrow D_MAX$
6 **end if**
7 **else** $R \leftarrow artificially_expanded_radius$
8 **end if**
9 **return** R

We also note that situations will occur when true clusters cover a larger area than the one *LeaderKernel* covers with D_MAX. In this case, different situations can be presented depended on the size of D_MAX and the mass we detect. The coding above show some of those situations. We leave the structure visible (for understandability purposes and for easy future adjustment) even though we return D_MAX as maximum radio. Still, expansion capabilities will be given, not coming from the *LeaderKernel* structure itself but from *StreamLeader* algorithm. We will cover these situations in next sections.

2.4.11 Temporal relevance

As stated before, we need automatic fast adaptation to change in underlying data distributions, also known as *Concept Drift*, including creation, evolution and disappearance of clusters. In this section we deal with the temporal aspect of this problem, that is, we will give more importance to new data than older one.

Data (instances) is now encapsulated in the *LeaderKernels*, therefore we will use their abstractions instead. We will follow a similar approach as the one used in *Clustream* [AHWY03] giving temporal importance to the data contained in the *LeaderKernels*. We will do this assigning them mean[50] and standard deviation[51] of arriving timestamps of instances contained in them, assuming that arrival timestamps are normally distributed. In that way, we can use the resulting values as a reference for time relevance.

When a new instance arrives and creates a new *LeaderKernel*, then the time relevance will be the one reflected by the arrival timestamp of that single instance. If, on the other hand, a *LeaderKernel* already exists and the instance is assigned to it, then the time relevance will be updated with the new timestamp and the *LeaderKernel* will be more *active* (it will render a higher, newest or more recent mean). Opposed situations arise when no new instances are assigned to an existing *LeaderKernel*. Then their time relevance becomes older and further away from current time, and, eventually, *obsolete* or *out-of-date*. When that happens, then the *LeaderKernel* that represents the cluster (or the homogeneous group of instances) is removed.

[49] We should also bear in mind that D_MAX is given by the user and, apart from the flexibility given, it follows his guidelines in terms of what sort of structures the algorithm should discover.
[50] $\mu = E[X] \approx \frac{LST}{N}$
[51] $\sigma = \sqrt{E[X^2] - (E[X])^2} \approx \sqrt{\frac{SST}{N} - (\frac{LST}{N})^2}$

Below the pseudo-code reflecting the concepts stated above. First, we calculate the mean time:

FUNCTION: GET_μ_TIME
INPUT: $\emptyset$
OUTPUT: TS timestamp representing the arrival mean time of all instances contained in the *LeaderKernel*

1 $TS \leftarrow \frac{LST}{N}$
2 **return** TS

Second, the standard deviation:

FUNCTION: GET_σ_TIME
INPUT: $\emptyset$
OUTPUT: TS timestamp representing the standard deviation of the arrival timestamps of all instances contained in the *LeaderKernel*

1 $TS \leftarrow \sqrt{\frac{SST}{N} - (\frac{LST}{N})^2}$
2 **return** TS

Finally, temporal relevance for the entire *LeaderKernel* is calculated as the sum of the two when the *LeaderKernel* was not artificially expanded. Otherwise we only consider μ time[52]:

FUNCTION: $GET_RELEVANCE_STAMP$
INPUT: $\emptyset$
OUTPUT: TS temporal relevancy timestamp of the entire *LeaderKernel*

1 $MuTime \leftarrow$ {get μ Time}
2 $SigmaTime \leftarrow$ {get σ Time}
3 **if not** $is_artificially_expanded$ **then**
4 $TS \leftarrow MuTime + SigmaTime$
5 **else**
6 $TS \leftarrow MuTime$
7 **return** TS

We are now in the position to calculate the temporal relevance of each *LeaderKernel* and also dynamically update them with incoming instances. Lastly, we need to determine whether a *LeaderKernel* is *actively* receiving instances and representing the current data stream or otherwise is receiving no instances or not in sufficient numbers to be representing the stream. In that case, those *LeaderKernels* should be considered *out-of-date* or *obsolete* and be removed.

[52] We will see further details on *LeaderKernel* expansion when we explain the *StreamLeader*, but we observed that, under special circumstances occurring when merging several artificial *LeaderKernel*, σ could potentially take large values sending temporal relevance of the resulting *LeaderKernel* into the future. This effect is undesirable because it could potentially maintain alive *LeaderKernels* that do not receive any instance for a certain time (longer than usual). Usual meaning that in a a normal scenario they would be treated as obsolete and be removed.

In Figures 16, 17, 18, we can artificially visualize the concepts explained above where instances from six different true clusters arrive from the data stream in a time span of 700 time units. Six *LeaderKernels* ($C1$, $C2$, $C3$, $C4$, $C5$,$C6$) try to encapsulate them. The temporal relevancy $\mu + \sigma$ for each *LeaderKernels* is then approximated by mean and standard deviation (shown by a vertical dotted line) using the set of instances allocated to each. This is done by assuming normal Gaussian distribution on the timestamps of the instances at arrival time[53].

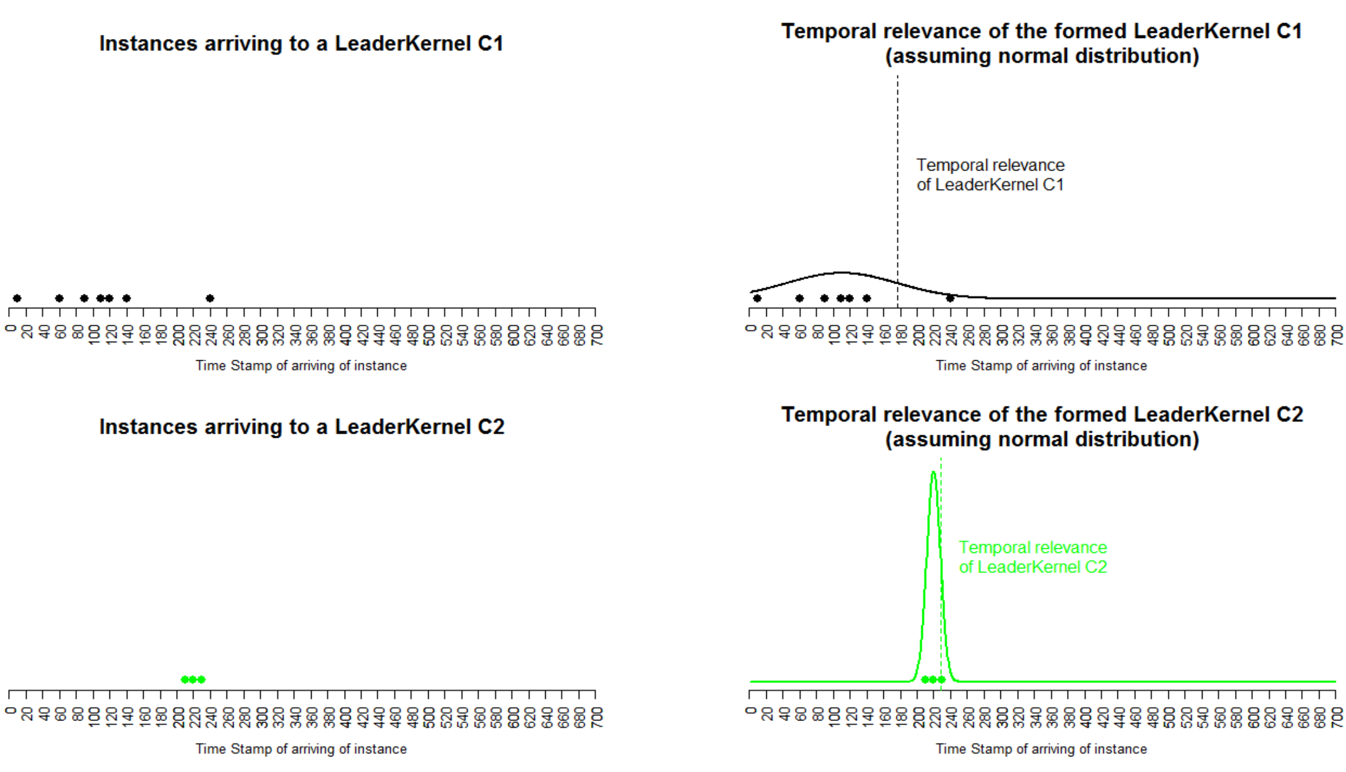


Figure 16: Temporal relevance of *LeaderKernel* as Gaussian $\mu + \sigma$ of timestamps of its instances (1)

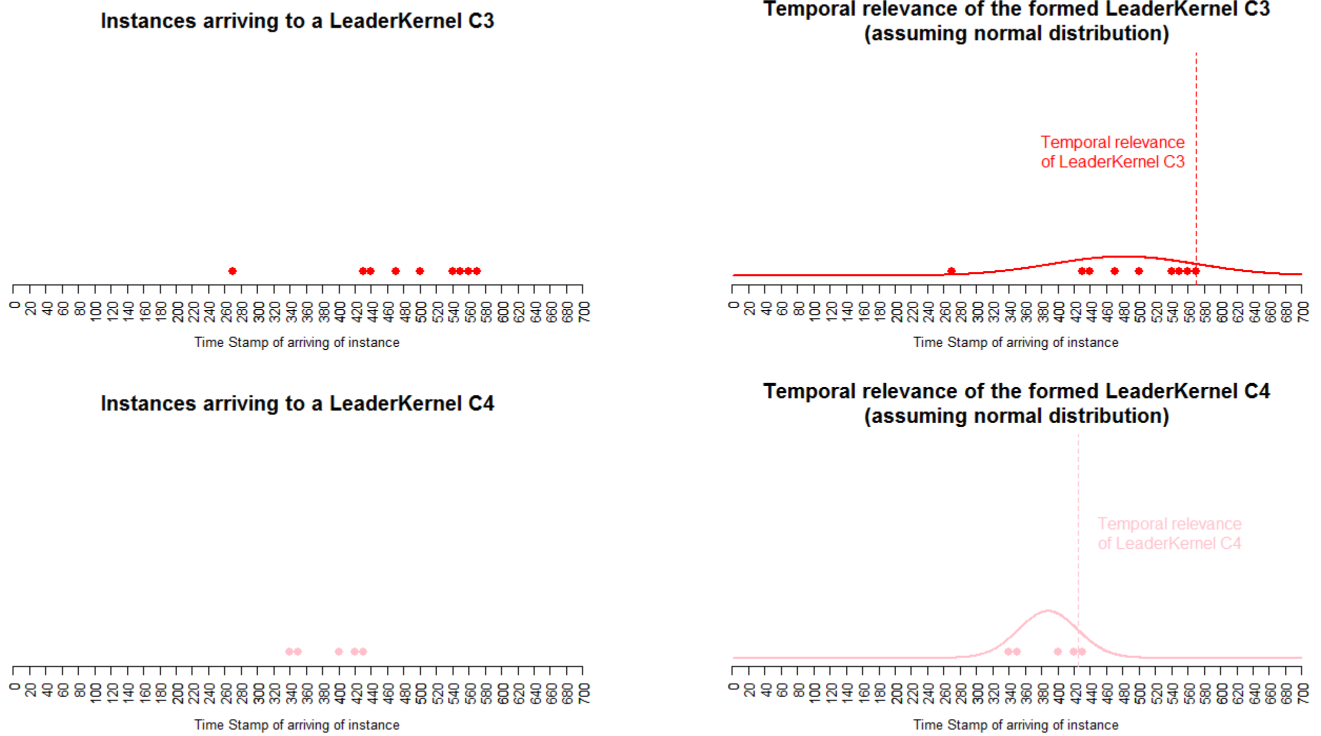


Figure 17: Temporal relevance of *LeaderKernel* as Gaussian $\mu + \sigma$ of timestamps of its instances (2)

[53]We assume no artificial expansion in these examples.

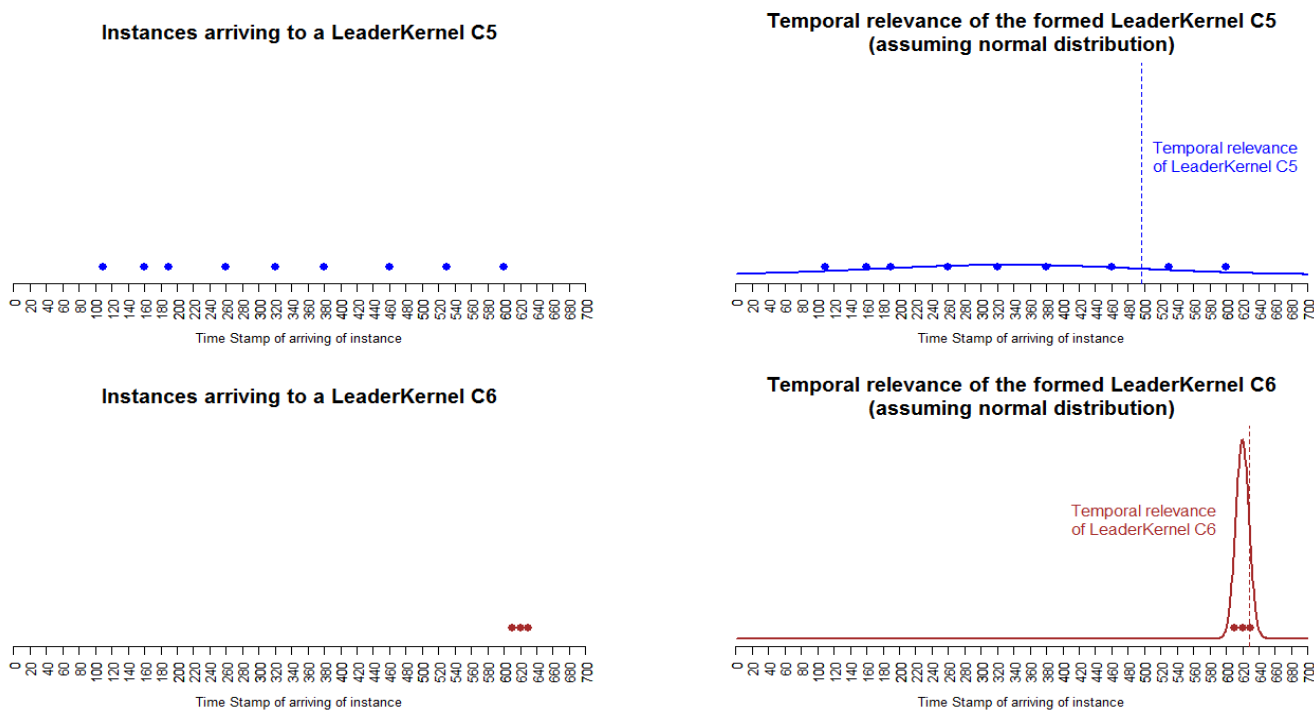


Figure 18: Temporal relevance of *LeaderKernel* as Gaussian $\mu + \sigma$ of timestamps of its instances (3)

Finally, we can bind together the concepts of temporal relevance and sliding window in order to keep only the *LeaderKernels* that are within the time window or *horizon*[54]. In this way, we are implementing the capabilities to handle the temporal aspect of *concept drift* where outdated *LeaderKernels* are removed and active ones kept and updated. Figure 19 shows a combined visual representation of the temporal relevancies (vertical lines) of the six *LeaderKernels* shown in 16, 17, 18. Using a *horizon* of size 600 time units, all *LeaderKernels* are considered as time relevant because their time relevancy falls within the *horizon*. Using a shorter *horizon* of size 250 implies that time relevancy for *LeaderKernels* 1, 2, 4 fall outside *horizon*, and are therefore not (time) relevant anymore, which means that they can be dropped.

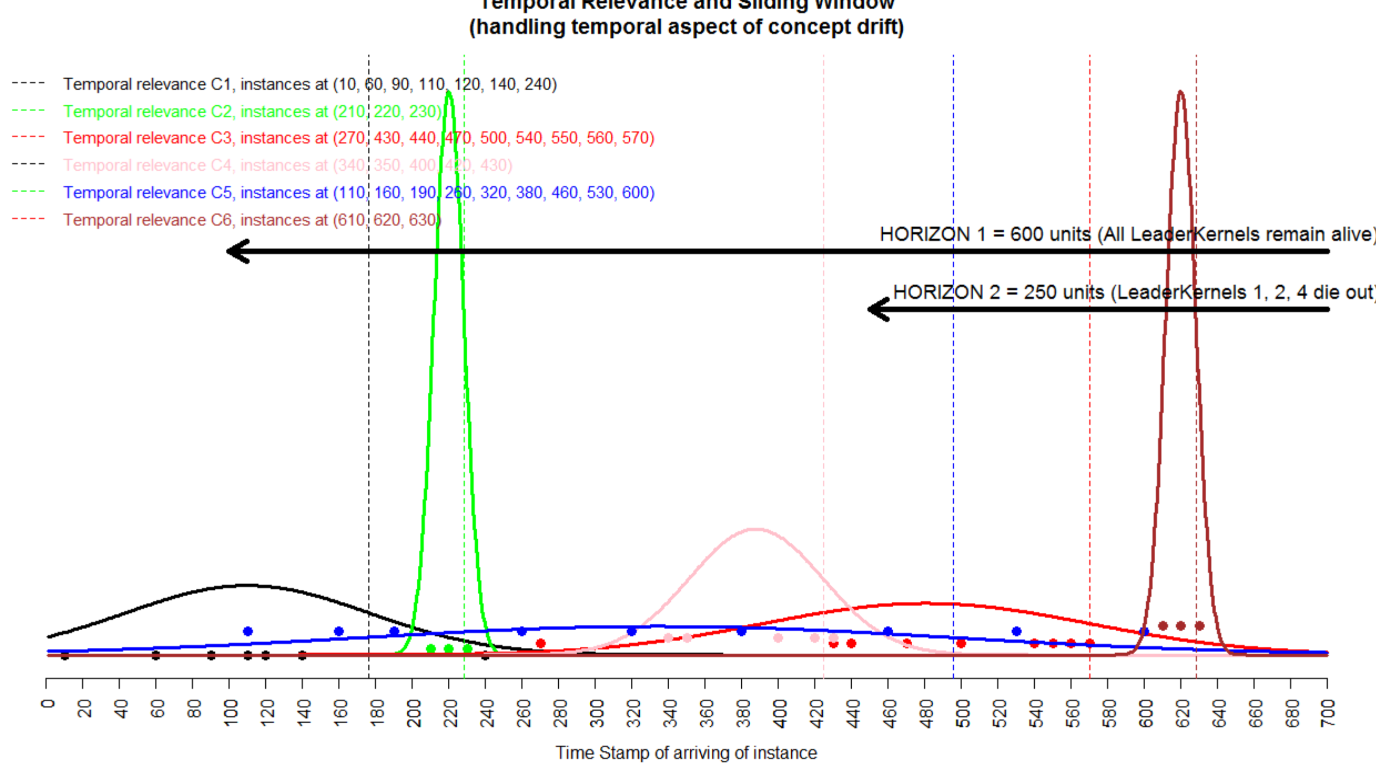


Figure 19: *LeaderKernels* are considered if their temporal relevance falls within *horizon*

[54]When time window models are used to handle *concept drift*, *Horizon* is always a user-defined parameter.

We observe that, depending on the time window, we keep or discard a different set of *LeaderKernels*. At time unit 700, with time window 1 (600 time units long), all *LeaderKernels* are considered as active since all their temporal relevances are within the time window. On the other hand, using time window 2 (250 time units long) means that temporal relevances for *LeaderKernels* 1, 2 and 4 fall behind the *horizon* threshold and are therefore phased out and discarded. We therefore verify what we saw in the theoretical aspects of data streaming: larger *horizons* will render more stability and less change. Smaller ones will make algorithms react much more quickly to changes in the stream (at the expense of more frequent processing[55])

2.4.12 Is same *LeaderKernel*

We can build an operation to distinguish the same two *LeaderKernels*:

FUNCTION: $IS_SAME_LeaderKernel$
INPUT: lk *LeaderKernel*
OUTPUT: I_S_LK boolean stating whether the *LeaderKernel* provided as input parameter is the same as this instance of the class

1 $N_lk \leftarrow$ {num instances contained in lk}
2 $LS_lk \leftarrow$ {LS in lk}
3 $SS_lk \leftarrow$ {SS in lk}
4 $LST_lk \leftarrow$ {LST contained in lk}
5 $SST_lk \leftarrow$ {SST contained in lk}
6 $LSD_lk \leftarrow$ {LSD contained in lk}
7 $creation_timestamp_lk \leftarrow$ {$creation_timestamp$ in lk}
8 $is_artificially_expanded_lk \leftarrow$ {check if lk was artificially expanded}
9 $radius \leftarrow$ {get radius}
10 $radius_lk \leftarrow$ {get radius from lk}
11 $I_S_LK \leftarrow (N = N_lk$ **and** $LS = LS_lk$ **and** $SS = SS_lk$ **and** $LST = LST_lk$ **and**
$SST = SST_lk$ **and** $LSD = LSD_lk$ **and**
$creation_timestamp = creation_timestamp_lk$ **and**
$is_artificially_expanded = is_artificially_expanded_lk$ **and** $radius = radius_lk)$
12 **return** I_S_LK

2.4.13 Inclusion probability

In order to calculate clustering quality metrics, MOA checks to which cluster (provided as output from the stream clustering algorithms) is an incoming instance allocated. This is then compared against the *ground-truth*[56] to check if the clustering algorithm clustered the instance properly. This is quite important since, as we will see in the *StreamLeader* workflow algorithm, D_MAX is always used as area of influence of a *LeaderKernel* for attracting new instances. The difference is that MOA will use the calculated radius, which might or might not match D_MAX, in order to check whether the output of the algorithm matches the ground-truth or the true clusters. This will be done by checking to which cluster is each instance allocated, and it will be possible by using the function described with the pseudo-code below:

The probability of an instance belonging to the *LeaderKernel*:

FUNCTION: $GET_INCLUSION_PROBABILITY$
INPUT: I stream data instance
OUTPUT: P probability of I belonging to this *LeaderKernel*

1 $L \leftarrow$ {*leader* of this *LeaderKernel*}
2 $R \leftarrow$ {radius of this *LeaderKernel*}
3 $D \leftarrow$ {$DIST_{SL}$ between L and I}
4 **if** $D \leq R$ **then return** 1
5 **else return** 0

[55] As we mentioned, techniques like *ADWIN* or *ADWIN2* in [BG06] could be very helpful to solve these trade-off situations.
[56] *Ground-truth* or *true clusters* are known by MOA when synthetic data is generated.

2.5 Special operations in *Offline* phase

StreamLeader will not use any batch clustering algorithm performed on the abstractions provided in the *online* phase to deliver final clustering. We tend to think that stream clustering algorithms should not resort to a conventional clustering algorithm to deliver clustering. We are of the opinion that doing so, we just simplify the problem of stream clustering to a problem of abstracting the stream and then just performing conventional clustering. Because of that, we decide to tackle the problem with a different approach. We will deliver final clustering by doing:

- Special noise treatment
- Expansion capabilities

2.5.1 Noise treatment 1: Percentile Cut

The conventional *leader* algorithm tends to either create new clusters (with new objects that are not similar enough to existing clusters) or integrate new points in exiting clusters (new points are similar enough). Extrapolated to the streaming scenario, the larger D_MAX, the fewer number of *LeaderKernels* with leaders will be created. The smaller D_MAX it is, the larger the number of *LeaderKernels* created.

We therefore need to control the number of *LeaderKernels* that will be created. We saw how the temporal relevance tackled effectively the notion of activeness or obsolescence for a *LeaderKernel* regarding the length of the *horizon*. Now we need to tackle the tendency of *StreamLeader* to create numerous clusters (*LeaderKernels*). This is because minimum amounts of noise levels away from true clusters would produce the effect of the creation of several *LeaderKernels* that reflect that noise. Most of the other algorithms we analyzed control this effect by numerous parametrization or trust the batch clustering algorithm executed in the *offline* phase with the task of disregarding that noise. We will follow a different approach independent from parametrization or conventional batch clustering. For the *StreamLeader*, the number of delivered clusters will be inversely proportional to D_MAX. The larger D_MAX the fewer amount of *LeaderKernels* and the smaller it is the larger the number. If the space is filled with *LeaderKernels*, we could assume that most of those *LeaderKernels* will be very sparsely populated compared with populated ones. That is, they will have very few instances as compared what true clusters would have. If we sort the *LeaderKernels* according to the instances they absorbed, in descendant order, we obtain a distribution with a tail on the right side, as shown in Figure 20. *LeaderKernels* located in that tail are likely absorbing only noise and are therefore the target of out percentile cut approach.

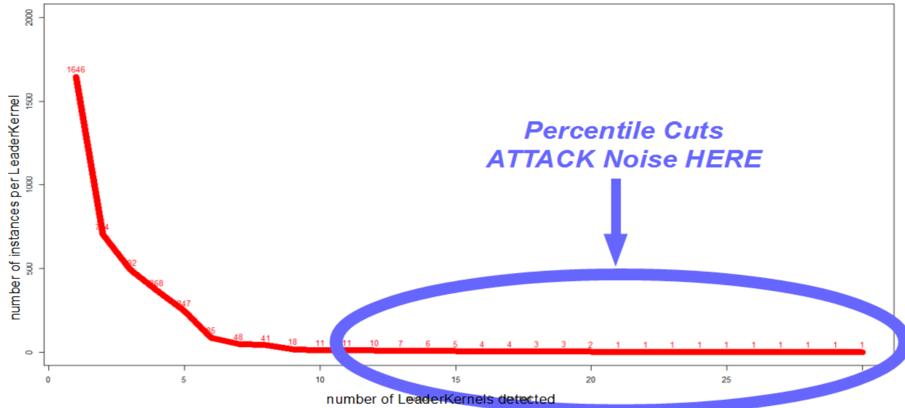


Figure 20: Percentile cut idea: attacking noise in the tail of a distribution of *LeaderKernels*

We can visualize the effect of MOA synthetic data generation with different sizes of D_MAX, applying 10% noise[57] and several data streaming configurations, to see how *LeaderKernels* are created and how many instances they absorb. We are particular interested in what sort of distributions we would find. Our aim is to eliminate noise aggressively and as early as possible. In the plots below we show the distribution of *LeaderKernels* according to instances contained and different percentile cuts based on the those instances. The values plotted using bigger font size (and matching colour with legend) show the true clusters with the real number of instances in them so that we can see which percentile cut would be most appropriate to get

[57] 0% noise would be most likely far too optimistic in any real data streaming environment.

rid of the noise.

Figure 21 shows *StreamLeader* running in MOA with a setting $D_MAX = 0.1$ at two random specific moments in the stream analysis. Around 30 *LeaderKernels* are maintained and a 70th cut seem to capture the true clusters properly:

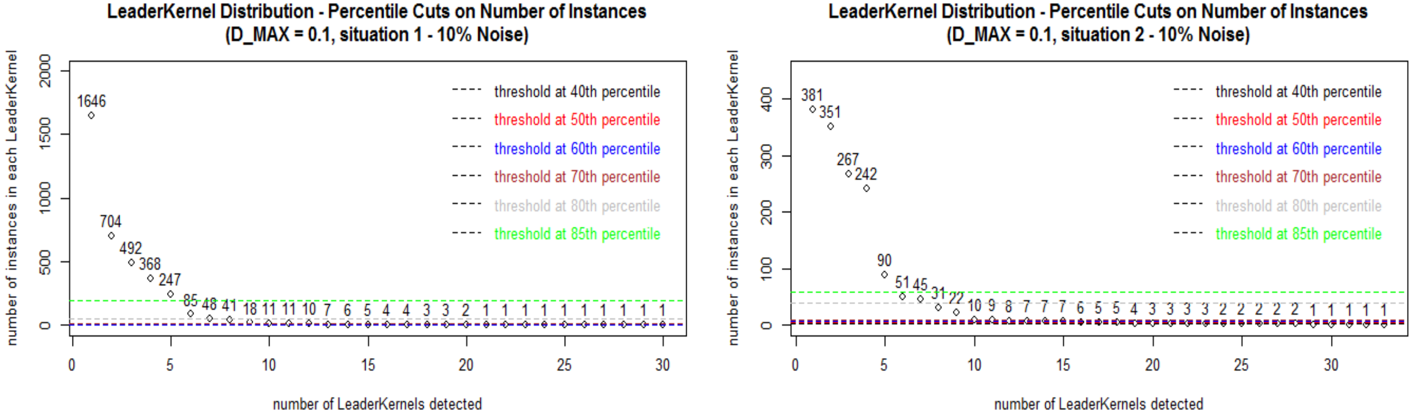


Figure 21: Distribution of *LeaderKernels* according to number of instances. Percentile cuts (1)

Figure 22 represents the same sort of experiments using $D_MAX = 0.2$ at four different instants. Now number of *LeaderKernels* is reduced to around 10 since the structures created are larger than those using $D_MAX = 0.1$. We see how percentile cuts behave in different distributions:

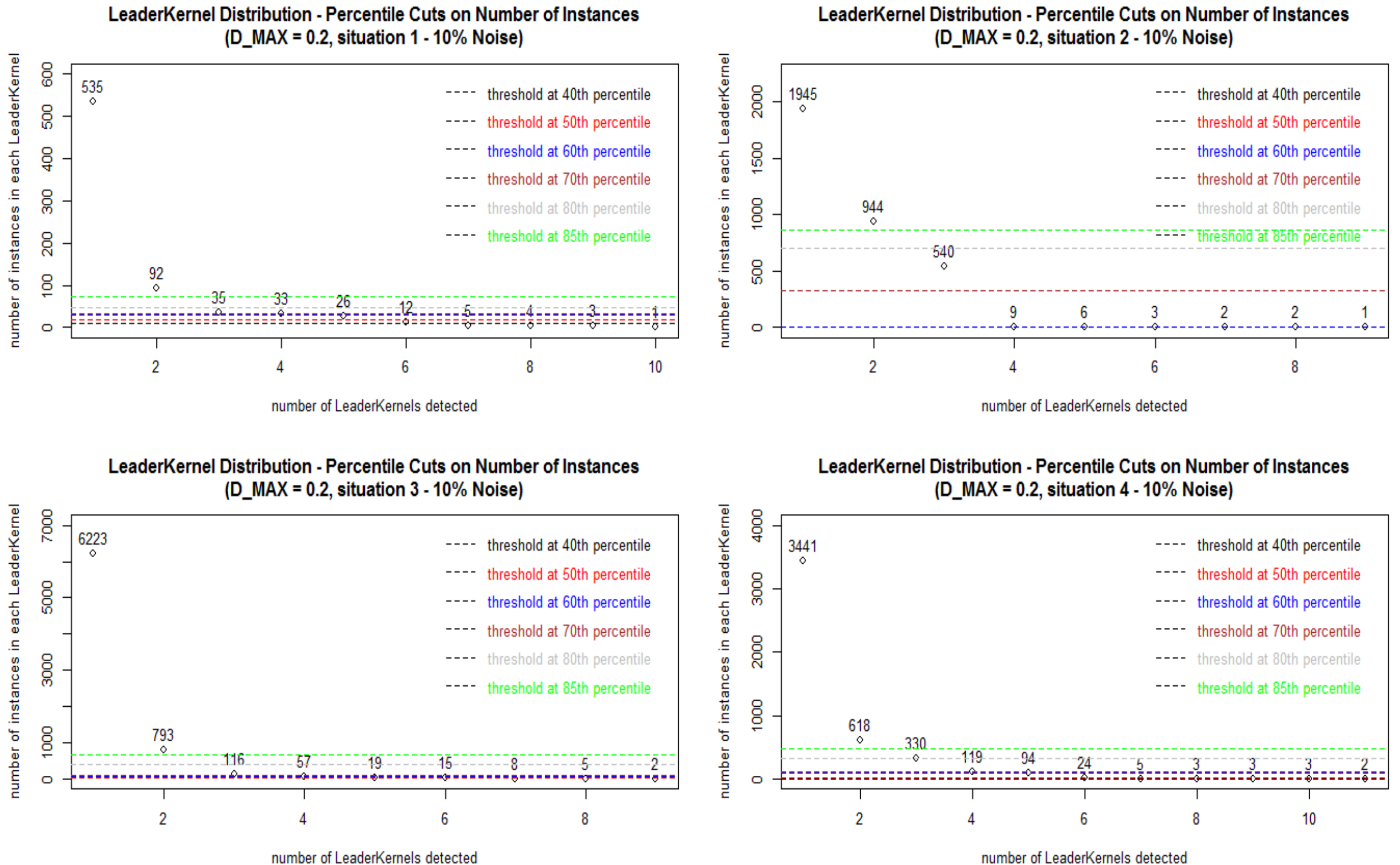


Figure 22: Distribution of *LeaderKernels* according to number of instances. Percentile cuts (2)

We observe that, if noise exists, it will be located always in the tail of the distribution. Because of its nature, the *leader* approach tends to create *LeaderKernels* capturing the mass of the true clusters and also the noise. We see that a specific percentile cut in each example would help reducing the true noise in the tail. In the examples above the proper cut to get rid of noise would range from 35th to 85th percentile.

While we used synthetic data generation from MOA for these examples, we noticed that the distribution of *LeaderKernels* based on weight (instances contained) follows an exponential-like distribution. This means that true clusters (and the *LeaderKernels* capturing them) contain most of the instances as compared *LeaderKernels* capturing only noise. In order to explore the flexibility of percentile cuts, we can imagine that the weight of *LeaderKernels* is, for instance, linear. Even if we can not generate this scenario in MOA, we manually create such distribution and visualize the results as shown in Figure 23:

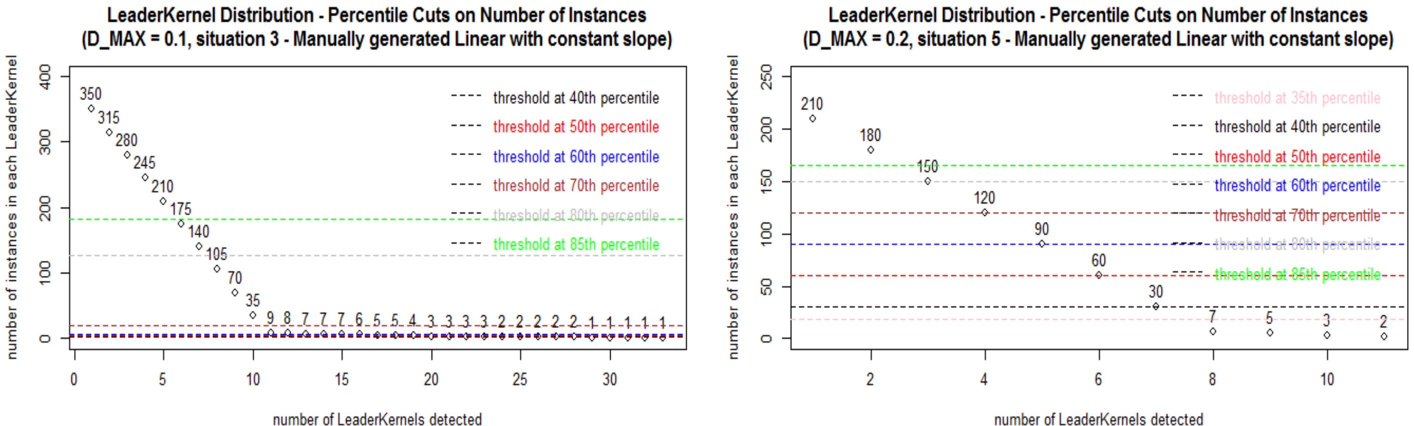


Figure 23: Manually generated distribution of *LeaderKernels*. Percentile cuts (3)

If assignation of instances to *LeaderKernels* was to be linear, percentile cuts would still be useful with more or less accuracy.

Lastly, we can think of the worst possible adverse scenario for the *StreamLeader* where the noise treatment could, at least, disregard the *LeaderKernel* with least instances. This scenario is a perfect 0% *noise-free* data stream environment. In Figure 24 a view of how the percentile cut approach would approximately work:

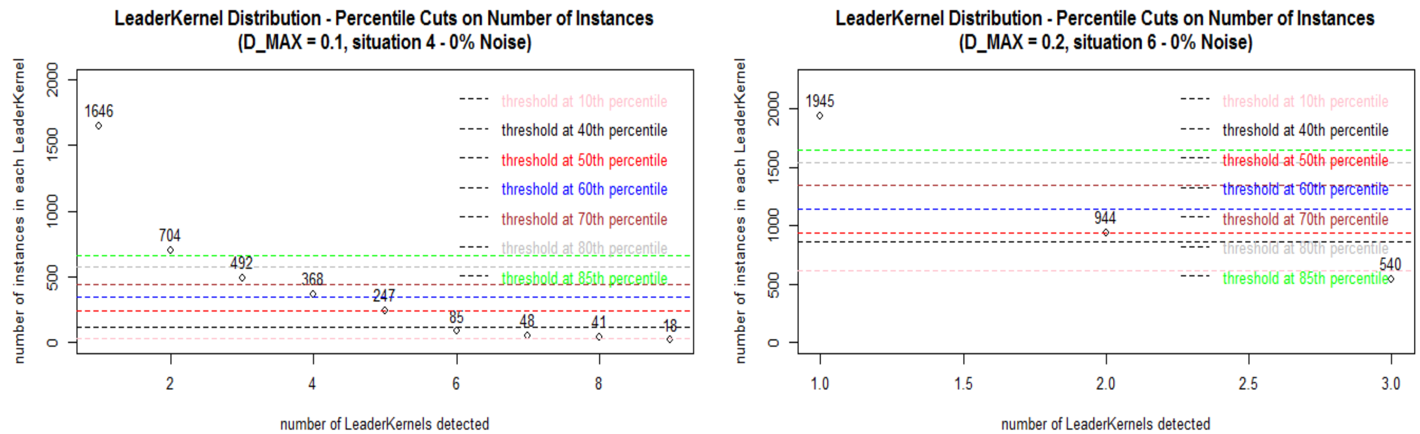


Figure 24: Distribution of *LeaderKernels* with 0% noise. Percentile cuts (4)

With exactly 0% noise, there would be no tail to be cut because there is no noise. Very low percentile cuts should be applied and at least, last *LeaderKernel* would be lost. In the left example above, the *LeaderKernel* with 18 instances would be discarded. In the right one, a *LeaderKernels* with 540 instances would be disregarded. This situation could occur but could be handled. Still, with just one instance appearing as noise, tail would exist again.

By knowing the behavior of percentile cuts based on the examples above, the uncertainty in noise levels (most likely above 0%) and the role the parameter D_MAX plays, we decide to be conservative, apply a 45% percentile cut to remove noise in the tail, and leave the rest of potential non-removed noise to the second noise treatment (explained in next section). We should not confuse a cutting point below 45th percentile in distribution of *LeaderKernels* with an stream containing 45% noise. The aim of the cut is to remove *LeaderKernels* that are capturing just noise in the stream, i.e. 5%, 20%, 40% or any. Whatever amount of noise there is, there will be always *LeaderKernels* capturing it (Hartigan's *Leader* concept will always fill the space with clusters as long as an instance appears where no other cluster already exists). Similarly *StreamLeader* will fill the *d*-space with *LeaderKernels* where noise appears. If noise is sparse in *d*-space, then many *LeaderKernels* will be created to fill those regions in space. On the other hand, if noise is very concentrated in a region (this now sounds close to what a real clusters is), fewer *LeaderKernels* will fill that particular region. In either of the two scenarios, several *LeaderKernels* with noise are created and most likely placed in the tail. Most likely scenario is to have many noisy *LeaderKernels*. A 45th percentile cut is flexible enough to remove many of them if necessary, but not as aggressive as to remove many proper *LeaderKernels* when the tail is not that long containing scattered noise. Worst case would be having noisy *LeaderKernels* having more weight than the real clusters themselves, but then it would not be noise, it would be a true cluster.

The part of the pseudo-code dealing with this noise treatment is as follows:

(L List of $LeaderKernels$)

53 $L \leftarrow \{Sort(L)$ in descending order according to weight of each $LeaderKernel\}$

55 $perc_threshold \leftarrow \{45$ percentile on distribution of L according to *LeaderKernel*'s weight$\}$
56 $\{$traverse $l \in L$ and remove l with $weight(l) < perc_threshold\}$

2.5.2 Noise treatment 2: Logarithmic Cut

Percentile cuts are useful for cutting tails in distributions and this is where noise would appear. However, situations could also occur when the cuts do not eliminate completely *LeaderKernels* with *few* instances which we would otherwise consider as noise. We therefore develop an additional strategy to get rid of potential noise that could be located on the elbow of the distribution, as Figure 25 shows.

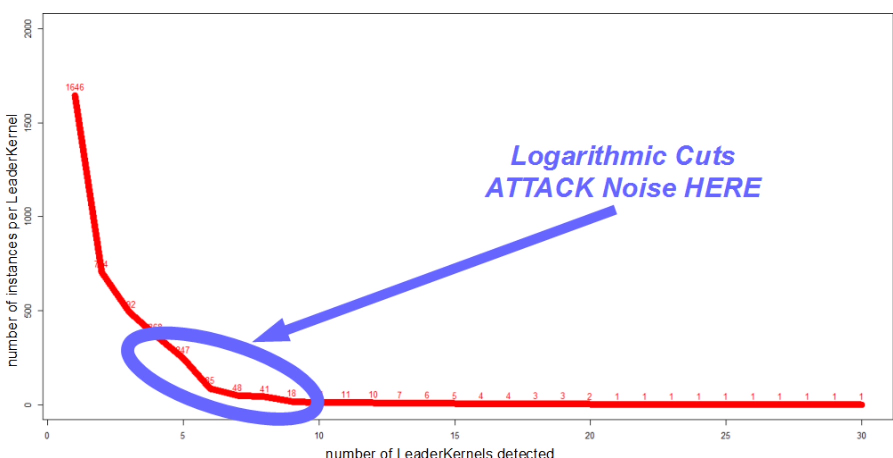


Figure 25: Logarithmic cut idea: attacking noise in the elbow of a distribution of *LeaderKernels*

We will basically try to estimate *on-the-fly* what *few* could be in the current data stream. We exploit again the sorted distribution of *LeaderKernels* and assume that, if there are still some in the tail representing noise, the amount of instances they contain would be in the around an order of magnitude smaller than the *LeaderKernels* representing the largest true cluster.

We show examples of scenarios with the situations described above. We apply percentile cuts to distributions based on *LeaderKernels* and finally several logarithmic cuts representing the concept of orders of magnitude. First two scenarios with $D_MAX = 0.1$ and 10% noise show in Figure 26. We can observe on the left plots, that numerous *LeaderKernels* are created with few instances (located in the tail). The plots in the center show the remaining distribution after applying a percentile cut, where a substantial number of noise has been eliminated. The plots on the right show the cut-off locations using different base logarithms. In both scenarios (upper and bottom plots) with many *LeaderKernels* in the distribution, a cut-off points base 1.1 logarithms would be too aggressive, eliminating *LeaderKernels* representing true clusters.

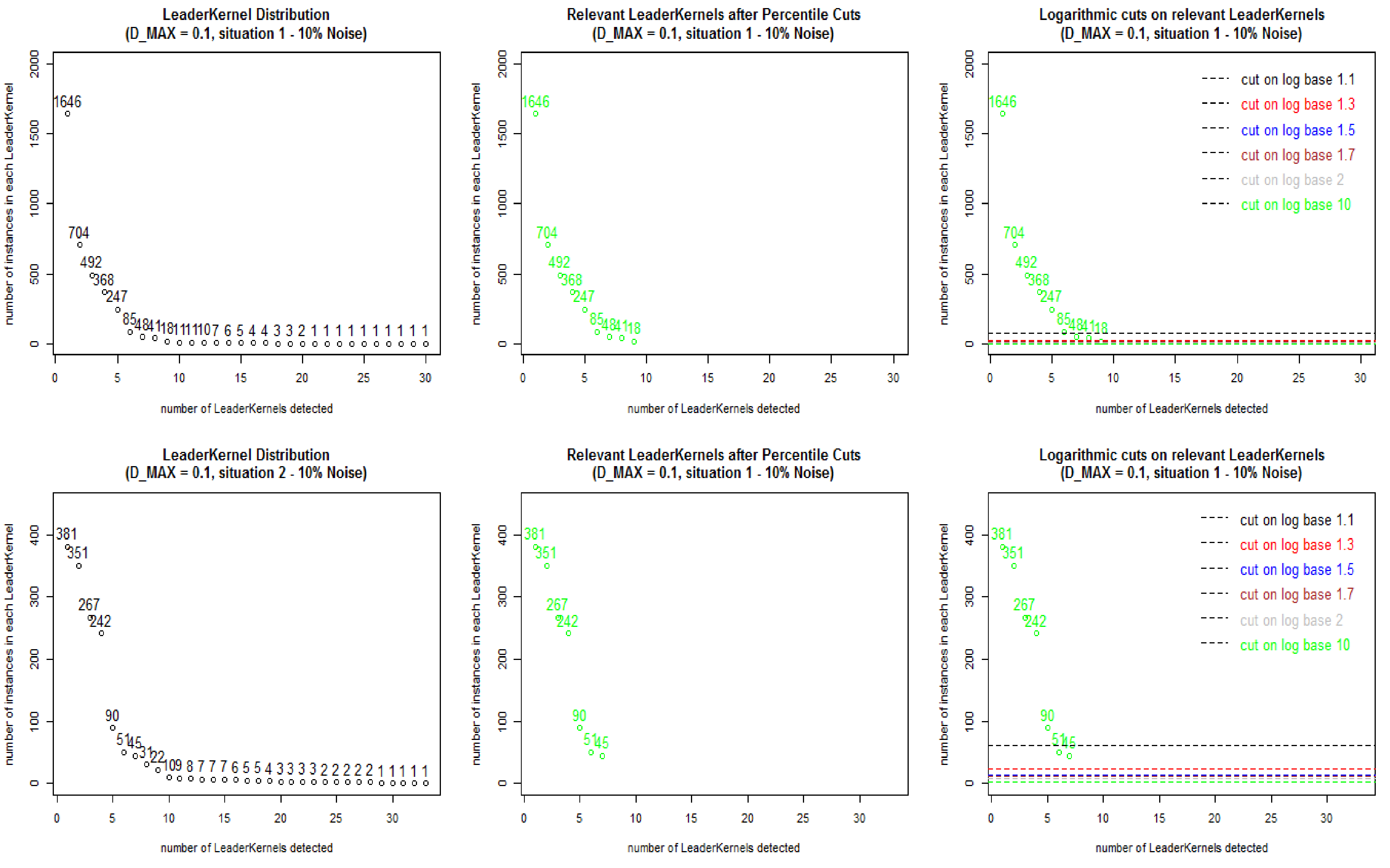


Figure 26: Logarithmic cuts on distribution of $LeaderKernels$ with $D_MAX = 0.1$, with 10% noise

Figure 27 shows *StreamLeader* with a setting of $D_MAX = 0.2$ and 10% noise, which implies a creation of fewer number of *LeaderKernels* than by using 0.1. With fewer clusters, we should be careful not to eliminate valid ones located in the elbow of the distribution. A conservative approach in these situations is therefore the choice of a lower base logarithm, since basis 1.1 is again too aggressive. Basis 1.3 seems conservative enough.

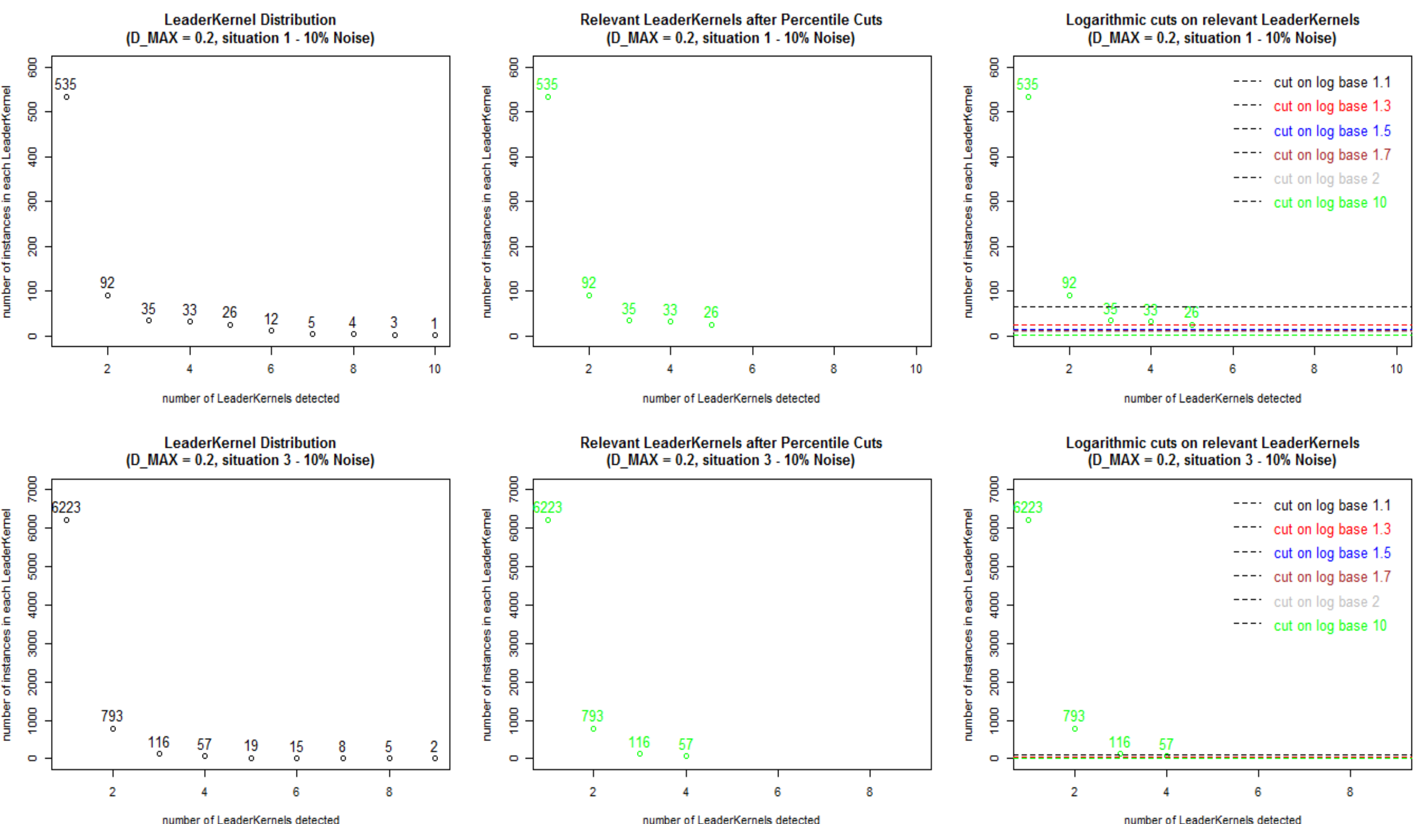


Figure 27: Logarithmic cuts on distribution of *LeaderKernels* with $D_MAX = 0.2$, 10% noise

As with percentile cuts, we would like to check logarithmic cuts in a distribution with constant slope (different from *exponential-like*). Since we can not create such scenario in MOA, we emulate manually the distribution. Figure 28 shows two of such situations. In the top plots, upper left shows a distribution of *LeaderKernels* based on weight. In the upper center, the distribution after performing 45th percentile cut. On the upper right, the different cut points in the distribution applying different logarithms. *Base 1.1* would be too aggressive by eliminating the *LeaderKernel* with 35 instances, which is a true cluster. Higher basis cut would not cut any of the true clusters. Bottom plots present another situation, with bottom left presenting a *LeaderKernel* with 70000 instances, a small set of true *LeaderKernels* 35 to 300 instances each, and lastly *LeaderKernels* capturing noise with 1 to 9 instances. Bottom middle pot shows the distribution after performing 45th percentile, which has eliminated noisy *LeaderKernels* of up to 35 instances weight. Bottom right shows where cuts would be executed using the different logarithms. Due to the magnitude of the 70000 weight *LeaderKernel* we see them all together. The cut-off values would be 117 for basis 1.1, 42 for basis 1.3, 27 for basis 1.5, 21 for basis 1.7, 5 for basis 10. Basis 1.1 is too aggressive, being basis 1.3 the one that removes noise better without eliminating *LeaderKernels* representing true clusters.

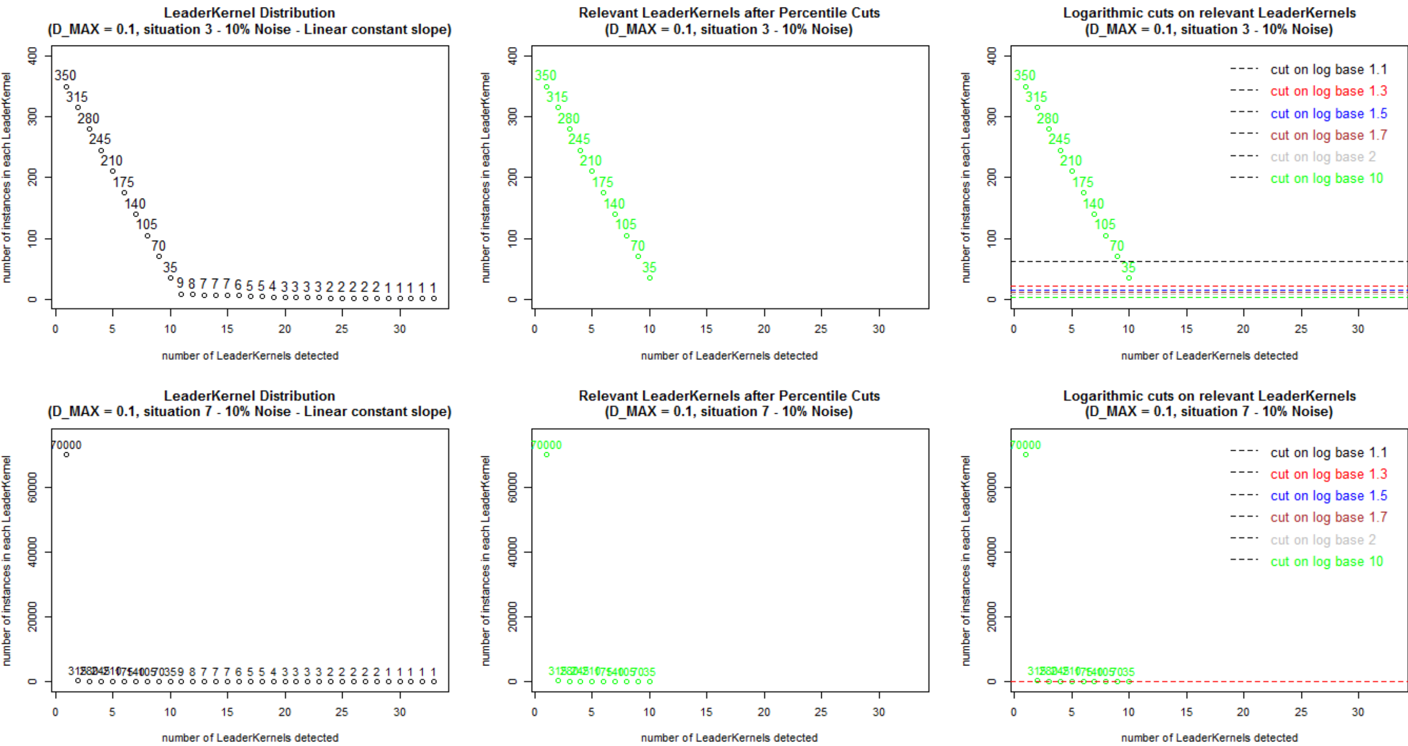


Figure 28: Logarithmic cuts on linear slope distribution of *LeaderKernels* with $D_MAX = 0.1$, 10% noise

With the situations described above, and having in mind that a stream will always be unexpected in behavior, we estimate that treating noise as if it was orders of magnitude smaller than true clusters is a reasonable solution. The aim is to use it as complementary technique to eliminate any remaining *LeaderKernel* that was not eliminated with the percentile cuts. From all the bases used in the scenarios, 1.3 seems a sensible choice, which eliminates noise and respects true clusters in most of the cases. Even with the large 70000 cluster and several small ones, only one true cluster with 35 instances would be incorrectly discarded.

As commented before, we are attempting to aggressive remove noise in a flexible way so we are not dependent on the use of conventional clustering and/or specialized parametrization. The two techniques combined and applied sequentially (percentile and logarithmic cut) make use of the distribution of *StreamLeaders* by attacking the noise in the tail first and then in the elbow of the distribution. They try to confer *StreamLeader* as much flexibility as possible to handle very diverse scenarios where noise and clusters could be presented in different ways, i.e. strong noise or weak, massive clusters only, combination of massive and sparsely populated plus noise and so on. The pseudo-code addressing noise though logarithmic cuts:

$(L \text{ List of } LeaderKernels)$

53 $\quad L \leftarrow \{Sort(L)$ in descending order according to weight of each $LeaderKernel\}$

57 $\quad log_biggest_lk \leftarrow \{$log base 1.3 of weight of first $LeaderKernel$ with more instances$\}$
58 $\quad \{$traverse $l \in L$ and remove l with $weight(l) < log_biggest_lk\}$

2.5.3 Expansion of intersecting *LeaderKernels* with radius D_MAX

D_MAX is the only parameter needed by the algorithm. It expresses the *guidelines* given by the user regarding how much proximity the instances should have in order to cluster them together (resulting in bigger or smaller clusters). We gave *LeaderKernels* the capability to contract according to the size of mass detected. This overrides D_MAX by adapting as much as possible to the structures perceived. However, there could also be situations where we want to override by expanding to a new cluster size that D_MAX can not fully capture. We will therefore allow the expansion of a *LeaderKernel* when we detect several *LeaderKernels* with maximum radius D_MAX that are overlapping. This is a good indication that the mass to capture requires a bigger radius.

In Figure 29, on the left side, we detect three *LeaderKernels* with radius D_MAX, being $\overline{LK_0}$ the one with largest mass or number of instances absorbed. Then we look for potential *LeaderKernels* with also radius D_MAX that overlap with $\overline{LK_0}$ by checking if distances [58] between respective leaders is $< (D_MAX * 2)$. In the middle, we observe how farthest distance to neighboring leaders is determined. This distance would indicate that, from leader of the *LeaderKernel* with most weight ($\overline{Leader_0}$) to the extreme of the farthest away *LeaderKernel* ($\overline{LK_1}$), the hyper-sphere with that distance as radius would most likely cover all overlapping *LeaderKernels* in space. On the right side, the *LeaderKernel* with largest mass will absorb its neighbors and will have its mass and statistical summarizations updated (Property 2, additivity), showing as ($\overline{LK_{0'}}$). It will be set as *artificially expanded*, and an new artificial radius will be approximated (125% $farthest_dist_between_leaders + D_MAX$), overriding D_MAX. 25% accounts for approximations due to the fact that we work always with summarized data. It is worth noticing that the leader $\overline{Leader_{0'}}$ representing $\overline{LK_{0'}}$ will not coincide with any of leaders $\overline{Leader_0}$, $\overline{Leader_1}$ or $\overline{Leader_2}$ that represented $\overline{LK_0}$, $\overline{LK_1}$ and $\overline{LK_2}$ respectively before the expansion process took place.

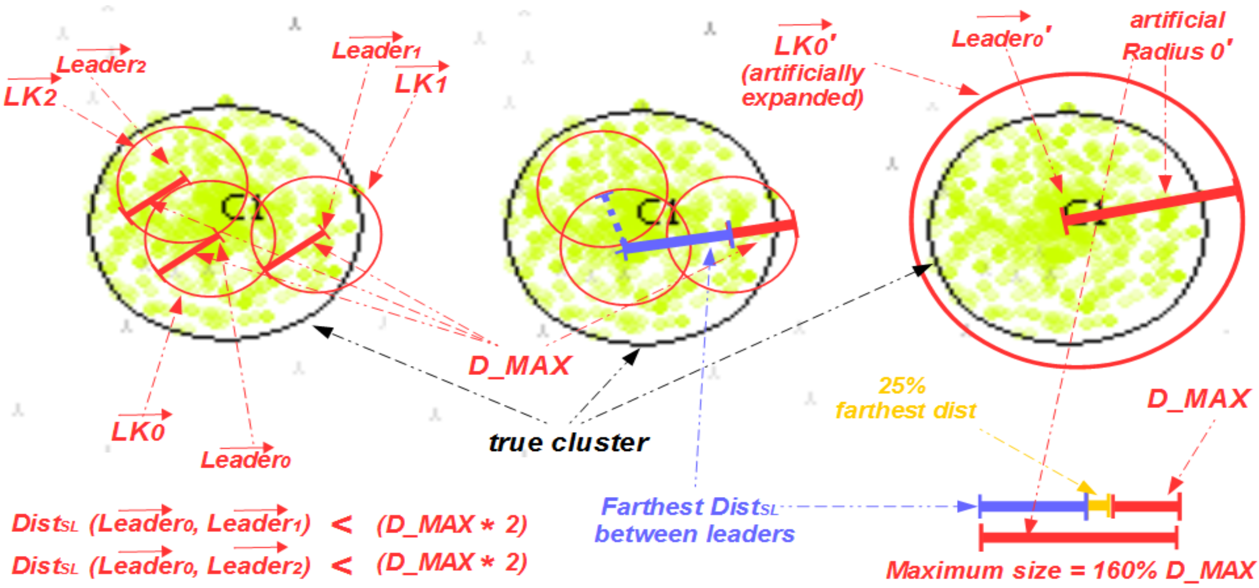


Figure 29: Intersecting *LeaderKernels* with radius D_MAX (left) merge into one bigger *LeaderKernel* (right)

[58] Distance function we defined in former sections.

A limit in the amount of growth is imposed. Since we work in normalized space and we still want to use user-defined D_MAX as a reference, we set an upper limit of 60% the value of original D_MAX. This should be seen as the amount of automatic flexibility we give to *StreamLeader* once D_MAX has been specified. It is of course possible that the cluster is still of bigger size. In that case, several *LeaderKernels* will try to capture its mass, but always limited to the 60% threshold[59].

The pseudo-code corresponding to this expansion is the following:

```
5    Expand_Factor_Max ← 60 percent
7    Radius_Expand_Max ← increase D_MAX by Expand_Factor_Max

59       lk_max ← {LeaderKernel radius D_MAX and largest weight}
60       lks_Engolf ← {LeaderKernels with radius D_MAX with leader at distance < (D_MAX * 2)
                 from leader of lk_max}
61       farthest_dist_between_leaders ← {max distance between leader of lk_max and leaders of lks_Engolf}
62       for lk_engolfed ∈ lks_Engolf do
63           lk_max ← lk_max ∪ lk_engolfed
64           L ← L − lk_engolfed
65       end for
66       if farthest_dist_between_leaders > 0 then
67           setArtifitiallyExpanded(lk_max)
68           newRadius ← 125% farthest_dist_between_leaders + D_MAX
69           if newRadius < Radius_Expand_Max then
70               setArtifitiallyExpandedRadius(lk_max, newRadius)
71           else
72               setArtifitiallyExpandedRadius(lk_max, Radius_Expand_Max)
73           end if
74       end if
```

2.5.4 Expansion of *LeaderKernels* with radius D_MAX

There could be also situations where a *LeaderKernel* with radius D_MAX can not capture fully a cluster but it senses that more mass is surrounding it in the d-dimensional space. Such situation is represented in Figure 30:

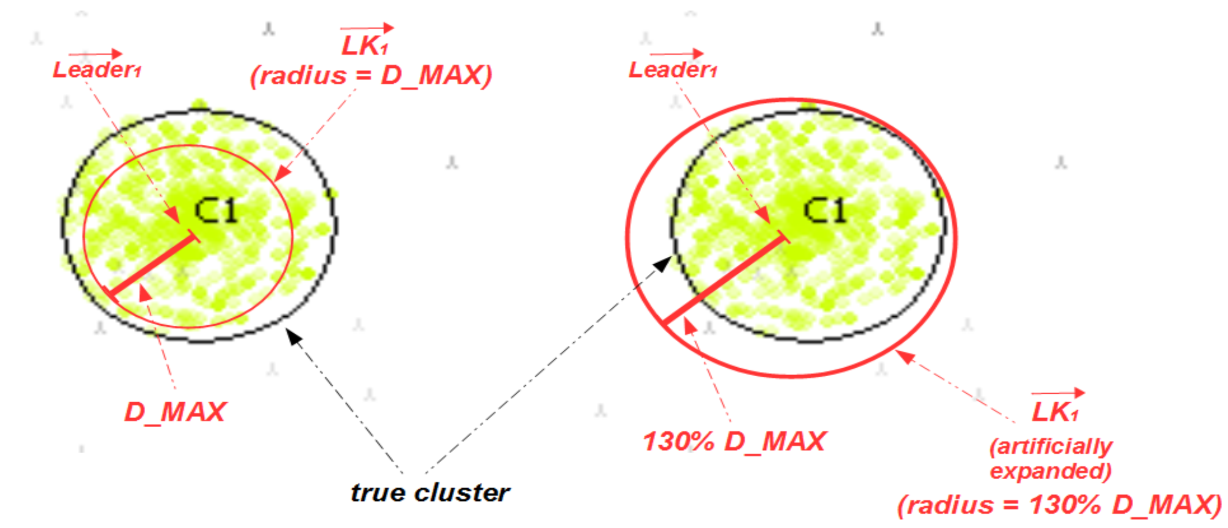


Figure 30: One (non overlapping) *LeaderKernel* with radius D_MAX (left) expands its radius (right)

We recall that *LeaderKernels* have the tendency to contract to smaller size when μ distance to leader of instances absorbed is less than 80% of half of D_MAX ($\frac{LSD}{N} < (\frac{D_MAX}{2} * 0.8)$). If we detect a *LeaderKernel*

[59] Additional expansion processes will be explained in next sections.

with maximum radius D_MAX, it means that it did not find the need to contract to a smaller appropriate radius. On the other hand, it is quite likely that more mass can be found in the surrounding space. The *LeaderKernel* is therefore given the chance to expand, not to the maximum allowed of 60% increase over D_MAX, but to an intermediate size of 30% increase. None of the summarized statistics are updated (including the leader $\overline{Leader_1}$), only the new radius is assigned and $\overline{LK_1}$ is set as *artificially expanded*. This will override the behavior of the *LeaderKernel* to return artificial radius and not D_MAX. It is also worth noticing that if such *LeaderKernel* is found, it is most likely isolated and not overlapping with others of D_MAX radius in space. Otherwise it would have merged with its neighbors as we explained in the first expansion method. Lastly, it is worth noticing that a situation could arise where a *LeaderKernel* was just created with automatic radius of size D_MAX and had no time to contract because it did not absorbed instances yet. In order to avoid that situation, we will expand only those with a creation date happening before $\frac{1}{4}th$ the duration of the *horizon* counting back from current time. We do this in order to make sure that they had some time units available to capture instances and had time to contract if needed.

Below the corresponding pseudo-code:

```
4   Expand_Factor_Min ← 30 percent
6   Radius_Expand_Min ← increase D_MAX by Expand_Factor_Min

75      expand_t_threshold ← timestamp − (H/4)
76      lks_ExtraRadius ← {LeaderKernel radius D_MAX and creation date < expand_t_threshold}
77      for lk_expand ∈ lks_ExtraRadius do
78          setArtifitiallyExpanded(lk_expand)
79          setArtifitiallyExpandedRadius(lk_expand, Radius_Expand_Min)
80      end for
```

2.5.5 Expansion of intersecting *LeaderKernels* with radius artificially expanded

There is a third situation where *LeaderKernels* might want to expand in size. The first two expansion processes took D_MAX *LeaderKernels* and produced another one with larger radius. In both cases, the resulting *LeaderKernel* was set as *artificially expanded* (meaning that user guideline D_MAX was overridden as radius). First process allowed a maximum expansion of 60% of original D_MAX, and, if not reaching that, to the estimated size according to the mass detected. The second process produced an expansion to a factor of 30% of original D_MAX, producing a *LeaderKernel* which is likely isolated in space and covers hopefully most of the true cluster. However, it is also possible that the second expansion was due to non-overlapping *LeaderKernels* that, together were trying to cover one unique cluster of much bigger size.

If the expansion succeeded, then the chances that overlapping now occurs are increased (otherwise the one *LeaderKernels* will remain isolated). If we detect it, then we proceed again with a merging process, allowing again to a maximum of 60% expansion ratio of original D_MAX.

Figure 31 shows in the left side three *LeaderKernels* with radio D_MAX, each covering a portion of d-dimensional space where high concentration of instances are located. They do not overlap, therefore, shown on the right, they are each artificially expanded (as explained in second type of expansion processes). A new radius of size 30% bigger than D_MAX is allocated to each one in the hope that they capture the cluster better. We should notice that when these expansions occur, the leaders do not change, only the radius are increased in sized (as opposed to merging where leaders are recalculated because of the new statistics of the involved *LeaderKernels*). The *LeaderKernel* with more weight (in this case $\overline{LK_2}$) and the overlapping neighbors ($\overline{LK_0}$ and $\overline{LK_1}$) are detected. This is done by checking the distances between pairs of leaders. If they overlap, then the distance is less than $((130\%\ D_MAX) * 2)$[60].

Finally, Figure 32 shows the last two steps to finish the expansion process. On the left, farthest distance between leaders is found. The one with more mass absorbs the others (by additive property of a *LeaderKernel*), the rest are discarded, and radius is calculated by adding the one they had to the farthest distance between leaders with a 25% increase fluctuation factor[61] to correct for approximations of a streaming envi-

[60] 30% increase over D_MAX is the amount of expansion flexibility we give to *StreamLeader* in the context of a normalized d-dimensional space and should not be mistaken with an input parameter.

[61] Again, this is an *informed* amount of flexibility we add in order to compensate for approximations in the streaming environment.

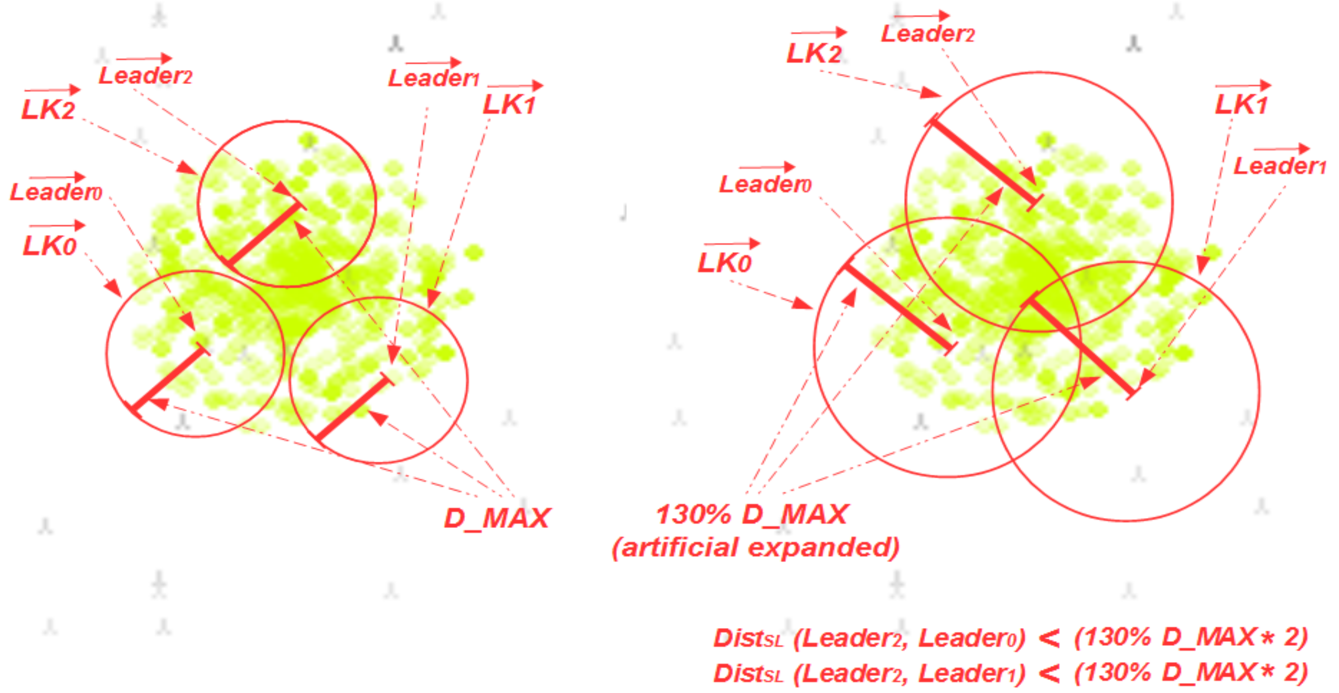


Figure 31: Three non overlapping *LeaderKernels* of radius D_MAX (left) expand radius and overlap (right) (1)

ronment. Again, if the new radius trespasses the 60% maximum expansion threshold, then that threshold is maintained. Otherwise the radio is the result of the calculation (125% $farthest_dist_between_radius + 130\%$ D_MAX). On the right, after the merging completes, the one *LeaderKernel* remains, its leader is placed in space according to the statistics of the combined *LeaderKernels*, and the new radius tries to cover more space than before the start of the process[62].

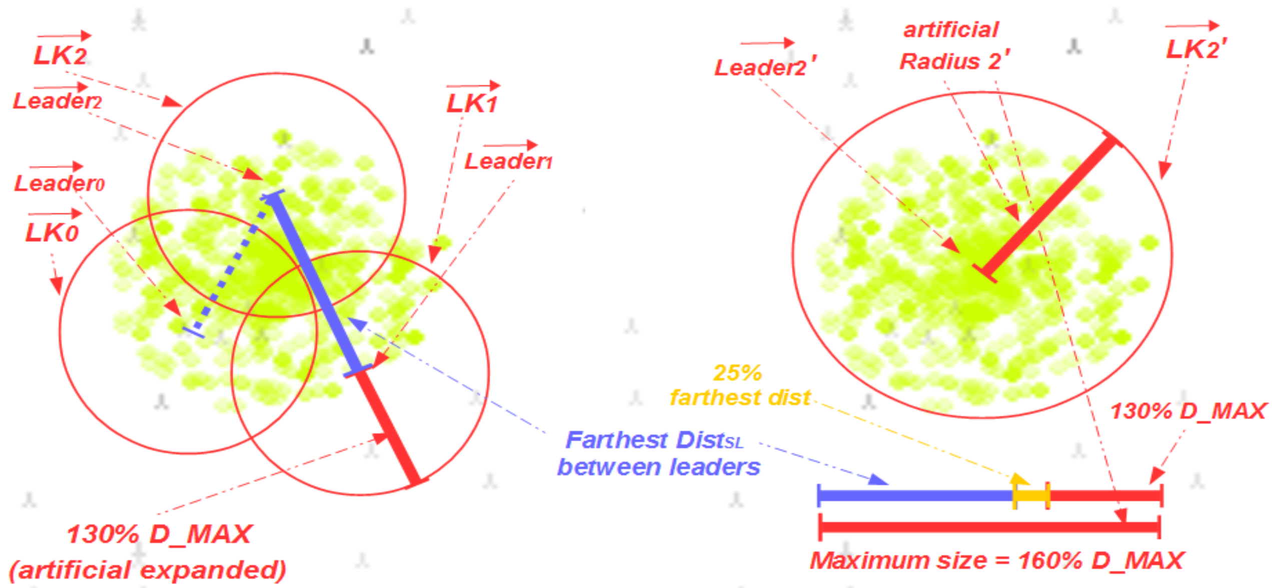


Figure 32: Three overlapping and expanded *LeaderKernels* (left) merge into a bigger one (right) (2)

[62] As we will see in the *StreamLeader* workflow algorithm, D_MAX is still used as area of influence for attracting new instances. The difference is that MOA will use this new calculated radius, and not D_MAX, in order to check whether the output of the algorithm matches the ground-truth or the true clusters. This will be done by checking to which cluster is each instance allocated, and it will be possible by using the $GET_INCLUSION_PROBABILITY$ function we defined for the *LeaderKernels*.

Below the corresponding pseudo-code:

```
4   Expand_Factor_Min ← 30 percent
5   Expand_Factor_Max ← 60 percent
6   Radius_Expand_Min ← increase D_MAX by Expand_Factor_Min
7   Radius_Expand_Max ← increase D_MAX by Expand_Factor_Max

81      lk_max ← {LeaderKernel with radius Radius_Expand_Min and largest weight}
82      lks_Engolf ← {LeaderKernels with radius Radius_Expand_Min
                with leader at distance < (Radius_Expand_Min * 2) from leader of lk_max}
83      farthest_dist_between_leaders ← {max distance between leader of lk_max and leaders of lks_Engolf}
84      for l_engolfed ∈ lks_Engolf do
85          lk_max ← lk_max ∪ l_engolfed
86          L ← L − l_engolfed
87      end for
88      if farthest_dist_between_radius > 0 then
89          setArtifitiallyExpanded(lk_max)
90          newRadius ← 125% farthest_dist_between_radius + Radius_Expand_Min
91          if newRadius < Radius_Expand_Max then
92              setArtifitiallyExpandedRadius(lk_max, newRadius)
93          else
94              setArtifitiallyExpandedRadius(lk_max, Radius_Expand_Max)
95          end if
96      end if
```

2.6 Pseudocode

We have already defined *LeaderKernels*, properties and operations as individual pieces. We will use them to generate statistical abstraction and adaptation to the dynamic data stream. On the other hand, we have also defined processes to carry out in the offline phase. They focus on aggressive noise treatment and expansion capabilities to describe better the discovered clusters, and they will be needed in order to compensate for not using any conventional clustering algorithm that take the abstractions as input (as most of other stream approaches do). In this section we can find first the proximity measure used. Then full pseudo-code related to *LeaderKernel*. Finally we will present our new stream clustering algorithm, called *StreamLeader*.

2.6.1 Proximity measure

DISTANCE FUNCTION:

FUNCTION: $DIST_{SL}$
INPUT: I^1, I^2 stream data instances in d-dimensional space (or any object in that same d-dimensional space, like a *leader* representing a *LeaderKernel*)
OUTPUT: D distance between the two given instances I^1, I^2

1 $d \leftarrow$ {dimensionality of I^1}
2 $D \leftarrow \sqrt{\sum_{i=1}^{d}(I_i^1 - I_i^2)^2}$
3 $D \leftarrow \frac{D}{D+1}$
4 **return** D

2.6.2 *LeaderKernel*

ATTRIBUTES:

LS: vector of d entries.
For each dimension, linear sum of all the instances added to the *LeaderKernel*. The p-th entry of LS is equal to $\sum_{j=1}^{n} x_p^j$.

- SS: vector of d entries.
For each dimension, squared sum of all the instances added to the *LeaderKernel*. The p-th entry of SS is equal to $\sum_{j=1}^{n}(x_p^j)^2$.

- LST: linear sum of the timestamps of all instances added to the *LeaderKernel*, equal to $\sum_{j=1}^{n} T^j$.

- SST: squared sum of the timestamps of all instances added to the *LeaderKernel*, equal to $\sum_{j=1}^{n}(T^j)^2$.

- LSD: linear sum of the distances to the leader of all instances added to the *LeaderKernel*, equal to $\sum_{j=1}^{n} D^j$.

- D_MAX: influence of the *LeaderKernel*, taken its leader as representative, in its neighborhood in the normalized d-dimensional space, measured in terms of the distance function $DIST_{SL}$, with range (0...0.5].

- N: Number of instances added to the *LeaderKernel*.

- *creation_timestamp*: timestamp that indicates when the *LeaderKernel* was created.

- *is_artificially_expanded*: boolean that indicates whether the *LeaderKernel* was artificially expanded farther than indicated by *D_MAX*.

- *artificially_expanded_radius*: radius of the *LeaderKernel* as a result of an artificial expansion process.

CONSTRUCTORS (INSTANCE & CLUSTER BASED):

PROCEDURE: $CREATE_{INSTANCE_BASED}$
INPUT: I stream data instance, d dimensions, TS timestamp for addition of I to *LeaderKernel*, $DMAX$ radio of influence of this *LeaderKernel*, $TSTMP_creation$ timestamp for creation of *LeaderKernel*
OUTPUT: $\emptyset$

1 $N \leftarrow 1$
2 $LS \leftarrow \{I$ on each dimension$\}$
3 $SS \leftarrow \{I^2$ on each dimension$\}$
4 $D_MAX \leftarrow DMAX$
5 $LST \leftarrow TS$
6 $SST \leftarrow TS^2$
7 $creation_timestamp \leftarrow TSTMP_creation$
8 $LSD \leftarrow 0$
9 $is_artificially_expanded \leftarrow$ false
10 $artificially_expanded_radius \leftarrow 0$

PROCEDURE: $CREATE_{CLUSTER_BASED}$
INPUT: lk *LeaderKernel*, $DMAX$ radio of influence around this *LeaderKernel*, $TSTMP_creation$ creation timestamp, AER artificially expanded radius assigned to *LeaderKernel*, WAE boolean indicating whether the *LeaderKernel* was artificially expanded farther than indicated by D_MAX
OUTPUT: $\emptyset$

1 $N \leftarrow \{$num instances contained in $lk\}$
2 $LS \leftarrow \{LS$ of $lk\}$
3 $SS \leftarrow \{SS$ of $lk\}$
4 $D_MAX \leftarrow DMAX$
5 $LST \leftarrow \{LST$ of $lk\}$
6 $SST \leftarrow \{SST$ of $lk\}$
7 $LSD \leftarrow \{LSD$ of $lk\}$
8 $creation_timestamp \leftarrow TSTMP_creation$
9 $is_artificially_expanded \leftarrow WAE$
10 $artificially_expanded_radius \leftarrow AER$

INSERT INSTANCE:

PROCEDURE: $INSERT_INSTANCE$
INPUT: I stream data instance, TS timestamp for addition of I to *LeaderKernel*, i_DTL distance from I to *leader* representing this *LeaderKernel*
OUTPUT: $\emptyset$

1 $N \leftarrow N + 1$
2 $LST \leftarrow LST + TS$
3 $SST \leftarrow SST + TS^2$
4 $LS \leftarrow \{LS + I$ on each dimension$\}$
5 $SS \leftarrow \{SS + I^2$ on each dimension$\}$
6 $LSD \leftarrow LSD + i_DTL$

GET LEADER:

FUNCTION: GET_LEADER
INPUT: $\emptyset$
OUTPUT: L as statistical center of the *LeaderKernel* in d-dimensional space

1 $L \leftarrow \{\frac{LS}{N}$ on each dimension$\}$
2 **return** L

MERGE (*LeaderKernels*):

PROCEDURE: $MERGE$
INPUT: lk_{add} *LeaderKernel* to be absorbed
OUTPUT: $\emptyset$

1 $radius \leftarrow$ {get radius}
2 $radius_lk \leftarrow$ {get radius from lk_{add}}
3 $creation_timestamp_lk \leftarrow$ {get creation timestamp in lk_{add}}
4 $N_lk \leftarrow$ {num instances contained in lk_{add}}
5 $LS_lk \leftarrow \{LS$ in $lk_{add}\}$
6 $SS_lk \leftarrow \{SS$ in $lk_{add}\}$
7 $LST_lk \leftarrow \{LST$ contained in $lk_{add}\}$
8 $SST_lk \leftarrow \{SST$ contained in $lk_{add}\}$
9 $LSD_lk \leftarrow \{LSD$ contained in $lk_{add}\}$
10 $is_artificially_expanded_l \leftarrow$ {check if lk_{add} was artificially expanded}
11 $creation_timestamp \leftarrow$ {if $N \neq 1$, $creation_timestamp$, otherwise $creation_timestamp_lk$ }
12 $N \leftarrow N + N_lk$
13 $LST \leftarrow \frac{(N*LST)+(N_lk*LST_lk)}{(N+N_lk)}$
14 $SST \leftarrow \frac{(N*SST)+(N_lk*SST_lk)}{(N+N_lk)}$
15 $LSD \leftarrow (N * LSD) + (N_lk * LSD_lk)$
16 $is_artificially_expanded \leftarrow is_artificially_expanded$ **or** $is_artificially_expanded_lk$
17 $artificially_expanded_radius \leftarrow$ {max between $radius$ and $radius_lk$}
18 $LS \leftarrow LS + LS_lk$
19 $SS \leftarrow \frac{(N*SS)+(N_lk*SS_lk)}{(N+N_lk)}$

HANDLING TIME:

FUNCTION: $GET_CREATION_TIME$
INPUT: $\emptyset$
OUTPUT: TS creation timestamp

1 $TS \leftarrow creation_timestamp$
2 **return** TS

FUNCTION: GET_μ_TIME
INPUT: $\emptyset$
OUTPUT: TS timestamp representing the arrival mean time of all instances contained in the *LeaderKernel*

1 $TS \leftarrow \frac{LST}{N}$
2 **return** TS

FUNCTION: GET_σ_TIME
INPUT: $\emptyset$
OUTPUT: TS timestamp representing the standard deviation of the arrival timestamps of all instances contained in the *LeaderKernel*

1 $TS \leftarrow \sqrt{\frac{SST}{N} - (\frac{LST}{N})^2}$
2 **return** TS

FUNCTION: $GET_RELEVANCE_STAMP$
INPUT: $\emptyset$
OUTPUT: TS temporal relevancy timestamp of the entire *LeaderKernel*

1 $MuTime \leftarrow$ {get μ Time}
2 $SigmaTime \leftarrow$ {get σ Time}
3 **if not** $is_artificially_expanded$ **then**
4   $TS \leftarrow MuTime + SigmaTime$
5 **else**
6   $TS \leftarrow MuTime$
7 **return** TS

HANDLING DISTANCE:

FUNCTION: $GET_\mu_DISTANCE_TO_LEADER$
INPUT: $\emptyset$
OUTPUT: D average distance to the leader representing the *LeaderKernel* (of all distances of all instances to leader at the time they were added to the *LeaderKernel*)

1 $D \leftarrow \frac{LSD}{N}$
2 **return** D

FUNCTION: $GET_RELEVANT_DISTANCE_TO_LEADER$
INPUT: $\emptyset$
OUTPUT: D distance

1 $D \leftarrow$ {get μ Distance to leader}
2 **return** D

FUNCTION: GET_RADIUS
INPUT: $\emptyset$
OUTPUT: R radius of the *LeaderKernel*

1 $Mu_D \leftarrow$ {get μ distance to leader}
2 $R \leftarrow 0$
3 **if not** $is_artificially_expanded$ **then**
4 **if** $Mu_D < (\frac{D_MAX}{2} * 0.8)$ **then** $R \leftarrow Mu_D * 2$
5 **else** $R \leftarrow D_MAX$
6 **end if**
7 **else** $R \leftarrow artificially_expanded_radius$
8 **end if**
9 **return** R

ARTIFICIAL EXPANSION:

PROCEDURE: $SET_ARTIFICIALLY_EXPANDED$
INPUT: IAE boolean indicating whether the *LeaderKernel* is artificially expanded
OUTPUT: $\emptyset$

1 $is_artificially_expanded \leftarrow IAE$

PROCEDURE: $SET_ARTIFICIALLY_EXPANDED_RADIUS$
INPUT: *AE_Radius* artificially expanded radius to be assigned to this *LeaderKernel* (overrides D_MAX as radius)
OUTPUT: $\emptyset$

1 $artificially_expanded_radius \leftarrow AE_Radius$

INCLUSION PROBABILITY:

FUNCTION: $GET_INCLUSION_PROBABILITY$
INPUT: I stream data instance
OUTPUT: P probability of I belonging to this *LeaderKernel*

1 $L \leftarrow$ {*leader* of this *LeaderKernel*}
2 $R \leftarrow$ {radius of this *LeaderKernel*}
3 $D \leftarrow$ {$DIST_{SL}$ between L and I}
4 **if** $D \leq R$ **then return** 1
5 **else return** 0

***LIDERKERNEL* COMPARISON**:

FUNCTION: $IS_SAME_LeaderKernel$
INPUT: *lk LeaderKernel*
OUTPUT: I_S_LK boolean stating whether the *LeaderKernel* provided as input parameter is the same as this instance of the class

1 $N_lk \leftarrow$ {num instances contained in lk}
2 $LS_lk \leftarrow$ {LS in lk}
3 $SS_lk \leftarrow$ {SS in lk}
4 $LST_lk \leftarrow$ {LST contained in lk}
5 $SST_lk \leftarrow$ {SST contained in lk}
6 $LSD_lk \leftarrow$ {LSD contained in lk}
7 $creation_timestamp_lk \leftarrow$ {$creation_timestamp$ in lk}
8 $is_artificially_expanded_lk \leftarrow$ {check if lk was artificially expanded}
9 $radius \leftarrow$ {get radius}
10 $radius_lk \leftarrow$ {get radius from lk}
11 $I_S_LK \leftarrow (N = N_lk$ **and** $LS = LS_lk$ **and** $SS = SS_lk$ **and** $LST = LST_lk$ **and**
$SST = SST_lk$ **and** $LSD = LSD_lk$ **and**
$creation_timestamp = creation_timestamp_lk$ **and**
$is_artificially_expanded = is_artificially_expanded_lk$ **and** $radius = radius_lk)$
12 **return** I_S_LK

ACCESS ATTRIBUTES:

FUNCTION: GET_LS
INPUT: $\emptyset$
OUTPUT: LS vector of d entries. For each dimension, linear sum of all the instances added to the *LeaderKernel*

1 **return** LS

FUNCTION: GET_SS
INPUT: $\emptyset$
OUTPUT: SS vector of d entries. For each dimension, squared sum of all the instances added to the *LeaderKernel*

1 **return** SS

FUNCTION: GET_LST
INPUT: $\emptyset$
OUTPUT: LST linear sum of the timestamps of all instances added to the *LeaderKernel*

1 **return** LST

FUNCTION: GET_SST
INPUT: $\emptyset$
OUTPUT: SST squared sum of the timestamps of all instances added to the *LeaderKernel*

1 **return** SST

FUNCTION: *GET_LSD*
INPUT: ∅
OUTPUT: *LSD* linear sum of the distances to the leader of all instances added to the *LeaderKernel*

1 **return** *LSD*

FUNCTION: *GET_DMAX*
INPUT: ∅
OUTPUT: *D_MAX* influence of the *LeaderKernel*, taken its leader as representative, in its neighborhood in the normalized d-dimensional space, measured in terms of the distance function $DIST_{SL}$, with range (0...0.5].

1 **return** *D_MAX*

FUNCTION: *GET_N*
INPUT: ∅
OUTPUT: *N* Number of instances added to the *LeaderKernel*
Observation: Because *LeaderKernel* extends *CFCluster* class in MOA, this function is also available as *GET_WEIGHT*.

1 **return** *N*

FUNCTION: *GET_CREATION_TIMESTAMP*
INPUT: ∅
OUTPUT: *creation_timestamp* timestamp that indicates when the *LeaderKernel* was created.

1 **return** *creation_timestamp*

FUNCTION: *IS_ARTIFICIALLY_EXPANDED*
INPUT: ∅
OUTPUT: *is_artificially_expanded*: boolean that indicates whether the *LeaderKernel* was artificially expanded farther than indicated by *D_MAX*.

1 **return** *is_artificially_expanded*

FUNCTION: *ARTIFICIALLY_EXPANDED_RADIUS*
INPUT: ∅
OUTPUT: *artificially_expanded_radius*: radius of the *LeaderKernel* as a result of an artificial expansion process.

1 **return** *artificially_expanded_radius*

2.6.3 *StreamLeader*

This section contains the pseudo-code describing the behavior of the *StreamLeader*, which is the complete stream clustering algorithm. It respects the general concept of the *leader* approach. We show first a flow chart with the summarization of the algorithm. Then a high-level version pseudo-code with the main tasks executed. Then a lower-level one with further details.

First, a flow chart with graphic icons associated with each task is shown in Figure 33:

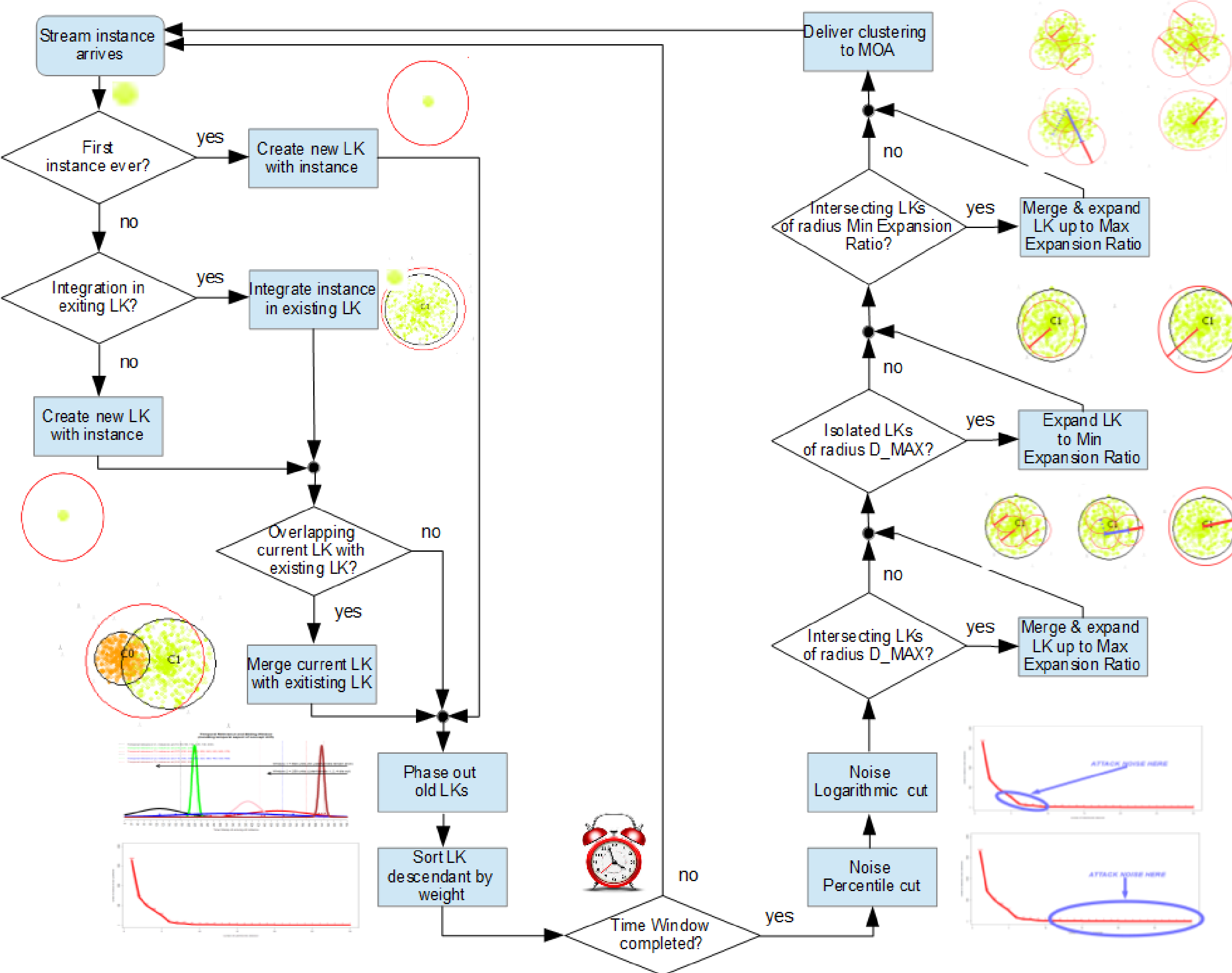


Figure 33: Flow chart of the *StreamLeader* algorithm

The algorithm requires one single parameter, D_MAX. It indicates the area of influence that a *leader*, as representative of its *LeaderKernel*, has in the multi-dimensional space [63]. We need to assign a default value that gives *StreamLeader* high flexibility overall, in any stream clustering scenario.

While reviewing the literature and the different algorithms, we find no homogeneous way in which default parametrization is chosen by the authors. Backing the choice with theoretical results does not seem a standard process. Instead, default parametrization is either advised by the authors or backed by experimental results obtained using specific data sets. Since a data stream is dynamic in nature and the data generation processes for those data sets can evolve, we prefer to base our choice not with experiments but with reasoning based on the key characteristics of *StreamLeader*. In that way, in order to decide on that value there are some factors to consider. *StreamLeader* works in d-dimensional space restricted to [0,1]. That means, biggest single hyper-spherical cluster covering maximum d space (reach max values in each dimension) would be $radius = 0.5$. We will not find that clustering problem normally, because almost every instance would be *similar* to any other, all falling in the same cluster. Two clusters with half of the maximum radius would be still considerably big size clusters, with $radius = 0.25$, which would be colliding in space as soon as there is a minimum of *concept drift* and movement of true clusters. Considering that *LeaderKernels* can contract to any smaller size, it is logical to think that we should make sure that its expansion capabilities are enough to capture a bigger (but not far bigger) true cluster of what D_MAX indicates. This is needed in order to avoid situations where many *LeaderKernels* try to capture a single true cluster.

A reasonable cluster size to cover using default parametrization would be around 70% of such cluster of $radius = 0.25$. That is, we need our *LeaderKernels* to be able to expand to a radius of $0.25 * 0.7 = 0.175$. Because *LeaderKernels* can expand a maximum factor of 60% of D_MAX, then *LeaderKernels* with $D_MAX = 0.11$ could reach to a radius $0.11 * 1.6 = 0.176$, which is the maximum radius size true cluster we want to capture properly (using just one *LeaderKernels*) using default parametrization, in d space [0,1]. To visualize the above mentioned cluster sizes in dimensions of size restricted to [0,1], we can see in Figure 34 a true cluster with maximum $radius = 0.5$ (left), two clusters of $radius = 0.25$ (middle) and finally a true cluster of $radius = 0.176$. Any cluster of radius smaller than 0.176 can be captured thanks to contraction capabilities.

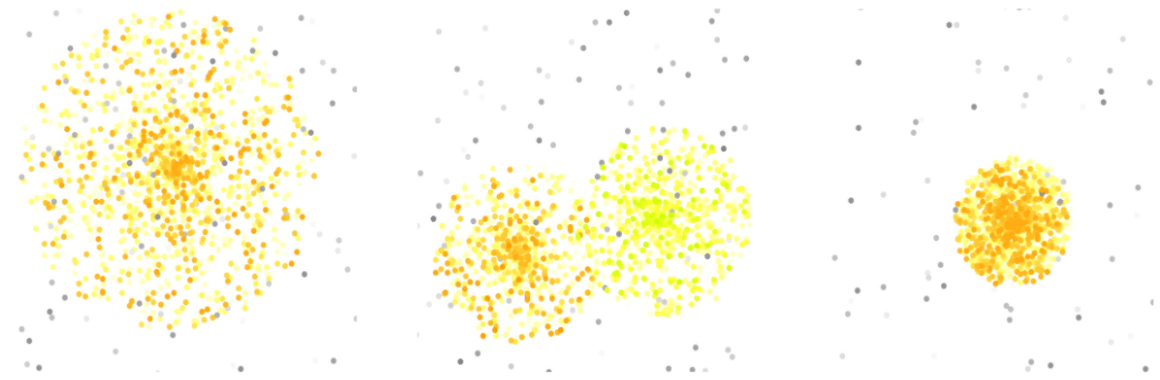


Figure 34: Cluster maximum radius 0.5 (left), two of radius $\frac{0.5}{2} = 0.25$ (middle) and one 70% $\frac{0.5}{2} = 0.176$

$D_MAX < 0.11$ it is also possible, but then we would have less changes of capturing bigger clusters with one unique *LeaderKernel*. Using several to detect a single true cluster would cause drops in quality measures like *Completeness*, *Rand Statistics*, etc. Figure 35 visualizes the problematic of covering a bigger true cluster of $radius = 0.176$ with several small *LeaderKernels* with $D_MAX = 0.06$.

[63] *Window Size* or *Horizon* is also a parameter, but not related to the algorithm but to the kind of analysis the user wants to carry out, in the sense that all stream clustering algorithms that use with time window models need it.

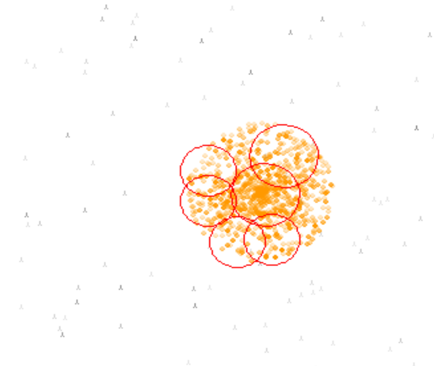


Figure 35: Clustering quality drops when a true cluster is covered by several smaller sized *LeaderKernels*

$D_MAX > 0.11$ it is possible as well, but then problems would arise in streaming scenarios where small size clusters are present. One single *LeaderKernel* would likely capture groups of several small true clusters, just because their masses fall within D_MAX range. This would penalize again *Recall*, *Homogeneity*, and other measures. Figure 36 shows a group of smaller size true clusters, radius in range 0.015 to 0.45, being captured by a single *LeaderKernel* with larger $D_MAX = 0.13$.

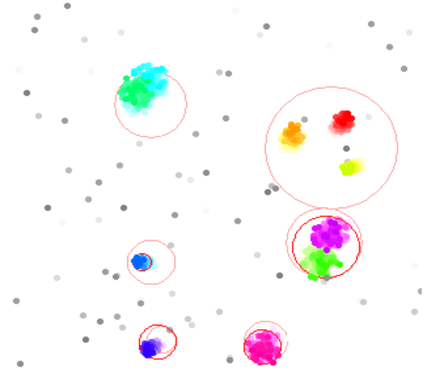


Figure 36: Clustering quality drops when several true clusters are captured by a single *LeaderKernels* with too large D_MAX

So $D_MAX = 0.11$ is a trade-off, between ability to capture a bigger single true cluster with just one *LeaderKernel* using expansion capabilities (and avoiding the use of several to capture its mass) and capturing a single small true cluster with a single *LeaderKernel* (and avoiding capturing several with one oversized *LeaderKernel*). Again, D_MAX should be understood as a user *guideline* regarding the size of structures the user wants to detect, or how low or high-level granularity in the data we want to isolate. *StreamLeader* tries to adapt that guideline, within certain limits, to the reality of the data distributions.

HIGH-LEVEL PSEUDO-CODE

INPUT: H time window, D_MAX radio of influence of each leader in normalized space
OUTPUT: L List of $LeaderKernels$ describing the clusters in the data stream relevant in the *horizon*

```
1   Initialize variables
2   while DATA STREAM ACTIVE do
3       Take incoming instance
4       if First instance ever then
5           Create new LeaderKernel with instance
6       else
7           Try integrate instance in existing LeaderKernel
8           if No integration then
9               Create new LeaderKernel with instance
10          end if
11      end if
12      if Overlapping current LeaderKernel with existing one then
13          Merge current LeaderKernel with existing one
14      end if
15      Phase out old LeaderKernels
16      LeaderKernel Sorting in descendant order according to weight
17      if Current time window completes then
18          Percentile cut on distribution of LeaderKernels according to weight
19          Logarithmic cut on distribution of LeaderKernels according to weight
20          Merge intersecting LeaderKernels of radius D_MAX and expand up to maximum ratio
21          Expand LeaderKernels with radius D_MAX to minimum expansion ratio
22          Merge intersecting LeaderKernels expanded to minimum ratio and expand up to maximum
23          Deliver L to MOA for cluster evaluation
24      end if
25  end while
```

LOW-LEVEL PSEUDO-CODE

INPUT: H time window, D_MAX radio of influence of each leader in normalized space
OUTPUT: L List of $LeaderKernels$ describing the clusters in the data stream relevant in the *horizon*

1 $L \leftarrow \emptyset$ (INITIALIZATION OF VARIABLES)
2 $timestamp \leftarrow$ pre-start sequence at zero
3 $initialized_leader \leftarrow$ false
4 $Expand_Factor_Min \leftarrow$ 30 percent
5 $Expand_Factor_Max \leftarrow$ 60 percent
6 $Radius_Expand_Min \leftarrow$ increase D_MAX by $Expand_Factor_Min$
7 $Radius_Expand_Max \leftarrow$ increase D_MAX by $Expand_Factor_Max$
8 **while** instance $\neq \emptyset$ **do** (WHILE DATA STREAM ON)
9 $I \leftarrow$ instance
10 Increase $timestamp$ in one time unit
11 $dim \leftarrow$ Dimensionality of I
12 $P_instance \leftarrow no\ LeaderKernel\ found$
13 $noMerged \leftarrow$ true
14 $noPhasedOut \leftarrow$ true
(INSERTION OF FIRST INSTANCE EVER)
15 **if not** $initialized_leader$ **then**
16 $L \leftarrow \{L \cup$ new $LeaderKernel$ with I, radius D_MAX, and $timestamp\}$
17 $P_instance \leftarrow$ newly added $LeaderKernel$
18 $initialized_leader \leftarrow$ true
19 **end if**
(INTEGRATION OF INSTANCE IN EXISTING $LIDERKERNEL$)
20 $J \leftarrow 0$
21 **while** $l \in L$ and $P_instance = no\ LeaderKernel\ found$ **do**
22 $dist \leftarrow DIST_{SL}$ (I, *leader* of $LeaderKernel\ l$)
23 **if** $dist \leq D_MAX$ **then**
24 $P_instance \leftarrow J$
25 $l \leftarrow \{l \cup I$ at $timestamp$ at distance $dist$ $\}$
26 **end if**
27 increase J
28 **end while**
(CREATION OF NEW $LIDERKERNEL$)
29 **if** $P_instance = no\ LeaderKernel\ found$ **then**
30 $L \leftarrow \{L \cup$ new $LeaderKernel$ with I, radius D_MAX, and $timestamp\}$
31 $P_instance \leftarrow$ position of newly added $LeaderKernel$
32 **end if**
(MERGING OF CURRENT $LIDERKERNEL$ WITH EXISTING ONE)
33 **if** $P_instance \neq no\ LeaderKernel\ found$ **then**
34 **if** number *LeaderKernels* ≥ 2 **then**
35 $l_merge \leftarrow \{$*LeaderKernel* at position $P_instance\}$
36 **while** $l \in L$ and $noMerged$ and $l \neq l_merge$ **do**
37 $dist \leftarrow DIST_{SL}$ (*leader* of l_merge, *leader* of l)
38 **if** $dist \leq D_MAX$ **then**
39 $l_merge \leftarrow l_merge \cup l$
40 $L \leftarrow L - l$
41 $noMerged \leftarrow$ false
42 **end if**
43 **end while**
44 **end if**
45 **end if**
(PHASING OUT OF OLD $LIDERKERNELS$)
46 $time_threshold \leftarrow timestamp - H$
47 **while** $l \in L$ and $noPhasedOut$ **do**
48 **if** $TimeRelevance(l) < time_threshold$ **then**
49 $L \leftarrow L - l$
50 $noPhasedOut \leftarrow$ false
51 **end if**
52 **end while**

(SORTING OF $LIDERKERNEL$ ACCORDING TO WEIGHT)

53 $L \leftarrow \{Sort(L)$ in descending order according to weight of each $LeaderKernel\}$

54 **if** *timestamp module* $H = 0$ **then** (CURRENT TIME WINDOW COMPLETES)

(NOISE PERCENTILE CUT)

55 $perc_threshold \leftarrow \{45$ percentile on distribution of L according to *LeaderKernel*'s weight$\}$
56 $\{$traverse $l \in L$ and remove l with $weight(l) < perc_threshold\}$

(NOISE LOGARITHMIC CUT)

57 $log_biggest_lk \leftarrow \{$log base 1.3 of weight of first $LeaderKernel$ with more instances$\}$
58 $\{$traverse $l \in L$ and remove l with $weight(l) < log_biggest_lk\}$

(EXPANSION OF INTERSECTING $LIDERKERNELS$ WITH RADIUS D_MAX)

59 $lk_max \leftarrow \{$*LeaderKernel* radius D_MAX and largest weight$\}$
60 $lks_Engolf \leftarrow \{$*LeaderKernels* with radius D_MAX with leader at distance $< (D_MAX * 2)$
from leader of $lk_max\}$
61 $farthest_dist_between_leaders \leftarrow \{$max distance between leader of lk_max and leaders of $lks_Engolf\}$
62 **for** $lk_engolfed \in lks_Engolf$ **do**
63 $lk_max \leftarrow lk_max \cup lk_engolfed$
64 $L \leftarrow L - lk_engolfed$
65 **end for**
66 **if** $farthest_dist_between_leaders > 0$ **then**
67 $setArtifitiallyExpanded(lk_max)$
68 $newRadius \leftarrow 125\%\ farthest_dist_between_leaders + D_MAX$
69 **if** $newRadius < Radius_Expand_Max$ **then**
70 $setArtifitiallyExpandedRadius(lk_max, newRadius)$
71 **else**
72 $setArtifitiallyExpandedRadius(lk_max, Radius_Expand_Max)$
73 **end if**
74 **end if**

(EXPANSION OF $LIDERKERNELS$ WITH RADIUS D_MAX)

75 $expand_t_threshold \leftarrow timestamp - (H/4)$
76 $lks_ExtraRadius \leftarrow \{$*LeaderKernel* radius D_MAX and creation date $< expand_t_threshold\}$
77 **for** $lk_expand \in lks_ExtraRadius$ **do**
78 $setArtifitiallyExpanded(lk_expand)$
79 $setArtifitiallyExpandedRadius(lk_expand, Radius_Expand_Min)$
80 **end for**

(EXPANSION OF INTERSECTING $LIDERKERNELS$ WITH RADIUS EXPANDED)

81 $lk_max \leftarrow \{$*LeaderKernel* with radius $Radius_Expand_Min$ and largest weight$\}$
82 $lks_Engolf \leftarrow \{$*LeaderKernels* with radius $Radius_Expand_Min$
with leader at distance $< (Radius_Expand_Min * 2)$ from leader of $lk_max\}$
83 $farthest_dist_between_leaders \leftarrow \{$max distance between leader of lk_max and leaders of $lks_Engolf\}$
84 **for** $l_engolfed \in lks_Engolf$ **do**
85 $lk_max \leftarrow lk_max \cup l_engolfed$
86 $L \leftarrow L - l_engolfed$
87 **end for**
88 **if** $farthest_dist_between_radius > 0$ **then**
89 $setArtifitiallyExpanded(lk_max)$
90 $newRadius \leftarrow 125\%\ farthest_dist_between_radius + Radius_Expand_Min$
91 **if** $newRadius < Radius_Expand_Max$ **then**
92 $setArtifitiallyExpandedRadius(lk_max, newRadius)$
93 **else**
94 $setArtifitiallyExpandedRadius(lk_max, Radius_Expand_Max)$
95 **end if**
96 **end if**
97 (Deliver L to MOA for cluster evaluation)
98 **end if**
99 **end while**
100 **return** L

3 Part 3 - Testing

Once *StreamLeader* is integrated successfully in MOA, we put it together with *Clustream*, *Denstream* and *Clustree* through an extensive test phase. First, we specify the computing resources available for testing. Second, we describe the metrics used for measuring quality in the results. Third, using MOA's synthetic data, we present the multiple test scenarios designed to test quality and sensibility for the four algorithms. Fourth, using also synthetic data, we will do extensive scalability test with different parametrization. Lastly, we will apply streaming real data and test quality results.

3.1 Computing resources used

We use a personal computer to run MOA and execute the tests. The technical specifications are the following:

- Processor: Intel(R) Core(TM) i7-3612QM CPU @ 2.10GHz
- RAM: 16.0 GB
- System type: 64 bits, x64
- Operative system: Windows 8.1, 2013 Microsoft Corporation

3.2 Quality metrics

Here we will discuss what metrics are available for measuring quality in clustering. Also, how other stream clustering algorithms were tested. Finally, we will decide which measures we will use fr testing.

With regards to the metrics available, they can be categorized in two sets: *internal* or *structural* and *external*. Internal measures only consider cluster properties, like cluster compactness, separation between clusters, distances between points within one cluster or between two different, etc. On the other hand, external measures, compare the resulting clustering provided by the algorithm against an already known true clustering, also known as *ground-truth*. This last approach takes into consideration factors like number of clusters, size, etc.

In general, most (if not all) of the internal and external measures were designed for static scenarios and have been used extensively in the last decades. *Silhouette index* [KR90], *Dunn's index* [Dunn74], *Sum of Square Distances SSQ* [HK01], *Root Mean Squared Standard Deviation RMSSTD* [Sha96] would be some internal measures used broadly. External measures include, among others, *Entropy* [SKK00], *Purity* [ZK04], *Completeness* [RH07], *Homogeneity* [RH07], *V-measure* [RH07], *Precision* [Rij79], *Recall* [Rij79], *F-measure* [Rij79], *F1-P* and F1-R in [MSE06], *Variation of Information* [Mei05], *Mutual Information* [CT06], *Rand statistic* [Rand71], *Jaccard coefficient* [FC83], *Minkowsky scores* [BSH+07], *classification error* [BSH+07], *van Dongen criterion* [Don00], *Goodman-Kruskal coefficient* [GK54], etc.

Because of the fact that streaming is a relatively new field of research, the above mentioned metrics have been also used in streaming scenarios. However, this new paradigm includes events like emerging, splitting or moving clusters, which pose a problem for conventional measures to capture them. Recently, a new evaluation measure for clustering on evolving data streams has been proposed in [KKJ+11], called *CMM*, *Cluster Mapping Measures*, which tackles the handling of such events. It is specifically designed for stream clustering by taking important properties of evolving data steams like cluster aging, cluster joining , diminishing, etc.

Regarding to how other stream clustering algorithms were tested in terms of quality, original publications of stream clustering algorithms show the use of SSQ and/or purity. This is the case of the three *StreamLeader*'s competitors in this project, namely *Clustream* [AHWY03], *Denstream* [CEQZ06] and *Clustree* [KABS11]. In streaming scenarios, SSQ takes the squared distances of the objects to the cluster centers. It has good reactions to cluster join and removal. However, it is not normalized, with unbounded upper limit, which poses obvious problems. Purity, on the other hand, is normalized in the [0,1] range, but it delivers fairly optimistic results when the number of clusters is large (achieving maximum purity when each point gets its own cluster). Therefore measuring clustering quality in scenarios of multiple clusters is not optimal.

Considering the above and metrics in MOA, we decide to take a set of seven quality measures, internal and external, and take their combined average. Since synthetic data generates labeled data, we will use preferentially external ones. Table 5 *CMM*, Table 6 *Rand Statistic*, Table 7 *Silhouette Coefficient*, Table 8 *Homogeneity*, Table 9 *Completeness*, Table 10 *F1-P* and Table 11 *F1-R* contain detailed descriptions of each of the seven metrics used. Table 12 Q_AVG contains the quality measure we will use for testing.

Quality Measure 1	**CMM** (Cluster Mapping Measures)
Type	External
Description	Cluster Mapping Measures are based on the concepts of connectivity between points and clusters, indicating how well a point fits the distribution of the cluster in comparison to the other points. It is a normalized sum of penalties, which includes missed points, misplaced points and noise inclusion. If no fault occurs, CMM returns 1 and 0 indicates maximum error. Misplaced points are calculated by mapping from clusters returned by stream clustering and ground-truth.
Formulation	$CMM(C,CL)^{64} = 1 - \frac{\sum_{x\in F} w(x)\cdot pen(x,C)}{\sum_{x\in F} w(x)\cdot con(x,Cl(x))}$ if $F=\emptyset$, then $CMM(C,CL)=1$ where $w(x)$: weight of an instance x $pen(x,C)$: overall penalties for fault points $con(x,Cl_i(x))$: point connectivity of a point x to a cluster C_i F: fault set, as set of objects mapped to a false class x: single instance from stream S formed by $\{x^1,...,x^N\}$ instances C: cluster set C CL: ground-truth clustering CL
Range	[0, 1]

Table 5: CMM measure details

Quality Measure 2	**Rand Statistic**
Type	External
Description	It measures the percentage of decisions that are correct, measuring similarity between classes and clustering delivered. It analyzes the agreements/disagreements of pairs or data points in different partitions. It gives equal weight to false positives and false negatives. Values close to 0 indicate that data clusters do not agree on any pair of points with the class and 1 indicating that they are exactly the same.
Formulation	$RI = \frac{TP+TN}{TP+FP+FN+TN}$ or $RI^{65} = \frac{[\binom{n}{2}-\sum_i\binom{n_i}{2}-\sum_j\binom{n_j}{2}+2\sum_{ij}\binom{n_{ij}}{2}]}{\binom{n}{2}}$ where TP: true positives TN: true negatives FP: false positives FN: false negatives n_i: number elements of class $c_i \in C$ n_j: number elements of cluster $k_j \in K$ n_{ij}: number elements of class $c_i \in C$ that are elements of cluster $k_j \in K$ n: number elements in data set
Range	[0,1]

Table 6: *Rand Statistic* quality measure details

[64] For full details on CMM measure please refer to [KKJ+11].

Quality Measure 3	**Silhouette Coefficient**
Type	Internal
Description	With ground-truth not being available, evaluation is done by the model itself. It is defined for each sample, comparing the average distances to points in the same cluster against the average distance to elements in other clusters. It uses combination of separation and cohesion measures, validating performance based on a pairwise difference of betweenness and within cluster-cluster distances. Score is higher when clusters are dense and well separated. It reflects cluster join errors when two classes are covered by a single cluster, but has problems covering other types of errors occurring in a stream. Also, scores could be higher for convex clusters than other concepts of clusters, such those returned by density-based algorithms.
Formulation	$SC^{66} = \frac{1}{NC} \sum_i \{ \frac{1}{n_i} \sum_{x \in C_i} \frac{b(x)-a(x)}{max[b(x),a(x)]} \}$ where $a(x)$: average dissimilarity of point x to cluster C_i, where smaller value means better assignment $b(x)$: lowest average dissimilarity of point x to any other cluster, of which x is not a member, where large value means bad matching with neighboring cluster NC: number of clusters C_i: i-th cluster n: number of objects in data set
Range	[-1,1], normalized in MOA to [0,1]

Table 7: *Silhouette Coefficient* quality measure details

Quality Measure 4	**Homogeneity**
Type	External
Description	Analogous to precision in the context of entropy based metrics. Perfect homogeneity is achieved when each cluster contains only elements of a single class. It can help in qualitative analysis to perceive what sort of mistakes are done in the assignments.
Formulation	$h^{67} = \begin{cases} 1 & \text{if} \quad H(C,K) = 0 \\ 1 - \frac{H(C\|K)}{H(C)} & \text{else} \end{cases}$ being $H(C) = -\sum_{c=1}^{\|C\|} \frac{\sum_{k=1}^{\|K\|} a_{ck}}{n} \log \frac{\sum_{k=1}^{\|K\|} a_{ck}}{n}$ $H(C\|K) = -\sum_{k=1}^{\|K\|} \sum_{c=1}^{\|C\|} \frac{a_{ck}}{N} \log \frac{a_{ck}}{\sum_{c=1}^{\|C\|} a_{ck}}$ where $H(C)$: entropy of the set of classes $H(C\|K)$: conditional entropy of the classes given the cluster assignments C: set of classes K: set of clusters $A = \{a_{ij}\}$: contingency table with clustering solution a_{ij}: number of data points members of class c_i and elements of cluster k_j N: number elements in data set n: number of classes
Range	[0, 1]

Table 8: *Homogeneity* quality measure details

[65] For full details on Rand Statistic measure, also called Rand Index, please refer to [Rand71].
[66] For full details on Silhouette Coefficient measure please refer to [KR90].
[67] For full details on Homogeneity measure please refer to [RH07].

Quality Measure 5	**Completeness**
Type	External
Description	Analogous to recall in the context of entropy based metrics. Perfect completeness is achieved when all members of a given lass are assigned to the same cluster. It can help in qualitative analysis to perceive what sort of mistakes are done in the assignments.
Formulation	$c^{68} = \begin{cases} 1 & \text{if} \quad H(K,C) = 0 \\ 1 - \frac{H(K\|C)}{H(K)} & \text{else} \end{cases}$ being $H(K) = -\sum_{k=1}^{\|K\|} \frac{\sum_{c=1}^{\|C\|} a_{ck}}{n} \log \frac{\sum_{c=1}^{\|C\|} a_{ck}}{n}$ $H(K\|C) = -\sum_{c=1}^{\|C\|} \sum_{k=1}^{\|K\|} \frac{a_{ck}}{N} \log \frac{a_{ck}}{\sum_{k=1}^{\|K\|} a_{ck}}$ where $H(K)$: entropy of the set of clusters $H(K\|C)$: conditional entropy of the clustering given class C: set of classes K: set of clusters $A = \{a_{ij}\}$: contingency table with clustering solution a_{ij}: number of data points members of class c_i and elements of cluster k_j N: number elements in data set n: number of classes
Range	[0, 1]

Table 9: *Completeness* quality measure details

Quality Measure 6	**F1-P**
Type	External
Description	Alternative method for conventional $F1$ measure (harmonic mean of precision and recall), proposed in [MSE06]. It calculates the total F1 clustering by maximizing its value for each found cluster.
Formulation	Steps for measure calculation are as follows[69]: 1) for each cluster found 2) find first best match 3) compute precision $P(c_i, k_j) = \frac{n_{ij}}{\|k_j\|}$ recall $R(c_i, k_j) = \frac{n_{ij}}{\|c_i\|}$ F measure $F(c_i, k_j) = \frac{2R(c_i,k_j)P(c_i,k_j)}{R(c_i,k_j)+P(c_i,k_j)}$ 4) overall average of all found clusters clustering F measure $F(C, K) = \sum_{k_j \in K} \frac{\|k_j\|}{N} \max \{F(c_i, k_j)\}$ where C: set of classes K: set of clusters n_{ij}: number elements of class $c_i \in C$ that are elements of cluster $k_j \in K$ N: number elements in data set
Range	[0, 1]

Table 10: *F1-P* quality measure details

[68] For full details on Completeness measure please refer to [RH07].

[69] For full details on F1-P measure please refer to [MSE06] and MOA's documentation. The formulation of F1-P could only be found in wording.

Quality Measure 7	**F1-R**
Type	External
Description	Alternative method for conventional $F1$ measure (harmonic mean of precision and recall), proposed in [MSE06]. It calculates the total F1 clustering by maximizing its value for each ground-truth class.
Formulation	Steps for measure calculation are as follows[70]: 1) for each cluster found 2) find first best match 3) compute precision $P(c_i, k_j) = \frac{n_{ij}}{\lvert k_j \rvert}$ recall $R(c_i, k_j) = \frac{n_{ij}}{\lvert c_i \rvert}$ F measure $F(c_i, k_j) = \frac{2R(c_i,k_j)P(c_i,k_j)}{R(c_i,k_j)+P(c_i,k_j)}$ 4) overall average of each ground-truth class clustering F measure $F(C, K) = \sum_{c_i \in C} \frac{\lvert c_i \rvert}{N} \max \{F(c_i, k_j)\}$ where C: set of classes K: set of clusters n_{ij}: number elements of class $c_i \in C$ that are elements of cluster $k_j \in K$ N: number elements in data set
Range	[0, 1]

Table 11: *F1-R* quality measure details

Finally, we define the quality measure we will use to do benchmarking:

Quality Measure used	**Q_AVG (AVG of Quality Metrics)**
Type	Combination of Internal & External
Description	Average of CMM, Rand Statistic, Silhouette Coefficient, Homogeneity, Completeness, F1-P and F1-R.
Formulation	$Q_AVG = \frac{CMM+RI+SC+h+c+F1_P+F1_R}{7}$ where CMM: CMM Cluster Mapping Measure RI: Rand Index SC: Silhouette Coefficient h: Homogeneity c: Completeness $F1_P$: F1-P $F1_R$: F1-P
Range	[0, 1]

Table 12: Q_AVG quality measure details

[70]For full details on F1-R measure please refer to [MSE06] and MOA's documentation. The formulation of F1-R could only be found in wording.

3.3 Quality tests - Synthetic data

Now that we have defined the quality measure to use ($Q_AVG = \frac{CMM+RI+SC+h+c+F1_P+F1_R}{7}$), we will need synthetic data for testing. We use MOA's capabilities to generate synthetic streaming data by using *Random RBF Generators*[71]. This gives us the flexibility to create very diverse scenarios and measure the quality results of the clustering delivered by *Clustream*, *Denstream* and *Clustree* in each. We should notice that generated data is normalized within the [0,1] range.

We therefore design a comprehensive set of scenarios where numbers of clusters (few, medium amount, many), radius sizes (small, medium size, big), dimensionality (2, 5, 20, 50) and noise levels (10%, 33%) are systematically combined. We produce 10 test type scenarios, combining all the elements above with the value ranges described in Table 13:

Number of clusters	- few: 2 to 5 - medium amount: 5 to 11 - many: 12 to 44
Cluster radius length	in d dimensions: - small: 0.001 to 0.05 - medium size: from 0.05 to 0.25[72] - big: 0.11 to 0.25 to - Observation: MOA generates synthetic data in normalized space. *StreamLeader* is designed to work in that space too.
Dimensionality	2, 5, 20, 50
Noise levels	10%, 32.8%

Table 13: Elements to create synthetic test scenarios

We prepare 10 different scenarios. Type 1 to 5 combine multiple cluster sizes with different amount of clusters, from low to high dimensionality, all in a 10% noise environment. Type 6 to 10 repeat the same sort of test scenarios but with noise levels increased to 33%. Each test in MOA will consist of 500000 streaming instances. It also contains *concept drift*, including cluster speed in the d space ($speed = 500$, clusters move a predefined distance of 0.01 every 500 instances), event frequency (each 50000 instances), events are either cluster creation, deletion, merging or split[73] when event frequency completes. Some parameters were taken as default, namely $decay_horizon = 1000$, $(instance) decay_threshold = 0.01$, $evaluation_frequency = 1000$. We used the given random seeds ($modelRandomSeed = 1$, $instanceRandomSeed = 5$).

With such tests, we aim to check how robust the algorithms are in terms of quality. We also want to check the sensibility in terms of parametrization. We opt to run each algorithm first with *default* then with *optimal* parametrization. Below we describe what we understand by these two different setups:

- **DEFAULT PARAMETRIZATION**: Parametrization MOA[74] gives by default to each algorithm. Values are as follows:

- *StreamLeader*: $horizon = 1000$, $d_max = 0.11$.

As stated before, a default value of 0.11 allows expansion that covers a big portion of the normalized d-dimensional space but also the contraction to discover small cluster.

- *Clustream*: $horizon = 1000$, $maxNumKernels = 100$, $kernelRadiFactor = 2$.

As we will see in optimal parametrization, this provides a good *micro-ratio* for up to 9 or 10 clusters.

- *Clustree*: $horizon = 1000$, $maxHeight = 8$.

[71] Please refer to Streaming Terminology section for further details.

[72] Medium size is generated creating clusters with radius ranging from 0.001 to 0.25.

[73] In some configurations, MOA crashed when splitting or creating new clusters, so we had to deactivate those options for those tests.

[74] *MOA Release 2014.11*

This height value for the tree is in line with what the authors mention in their published paper in [KABS11], where they use heights ranging from 7 to 11.

- *Denstream (with DBSCAN)*: *horizon* = 1000, *epsilon* = 0.02, *beta* = 0.2, *mu* = 1, *initPoints* = 1000, *offline* = 2, *lambda* = 0.25, *processingSpeed* = 100

We already see the heavy and complex parametrization the algorithm needs. In general, these need to be fine-tuned according to the data we are receiving. Parameters guide the algorithm in how data should be gathered within common areas of density or noise elimination, and they are used first by *Denstream* in the online phase and then by *DBSCAN* in the offline. Authors describe in [CEQZ06] that parameters should adopt the following setting: ϵ (neighbourhood) = 16 β (outlier threshold) = 0.2 μ (weight threshold for micro-cluster to exists as such) = 10 *InitPoints* = 1000 λ (decay factor) = 0.25 *processingSpeed* = 1000

We note that MOA's parametrization is different. So we use MOA's as we did with the other algorithms. We still can try to adjust the parameters when we do optimal parametrization.

- **OPTIMAL PARAMETRIZATION** as specified by corresponding published paper or adjusting the parameters as good as possible to the knowledge we have of the incoming stream (we know ground-truth because we specify the generating data distributions):

- *StreamLeader*: We adjust radius *d_max* to a size around 15% smaller than the radius of the biggest structure we will receive in the stream. Doing that, expansion capabilities will be used to cover the biggest cluster entirely. If at the same time several much smaller clusters do appear, we want that *d_max* as small as possible in order capture them individually and not together with bigger clusters that can cover several.

- *Clustream*: As described in [AHWY03], in order to achieve quality clustering the algorithm needs that the number of micro-clusters is larger than the one for natural clusters. That that can be inefficient to maintain if the number increases enough. Therefore a *micro-ratio* is defined as $\frac{number_micro-clusters}{number_natural_clusters}$ with a value of at least 10 or above to provide quality results with enough low granularity. *KernelRadiFactor* = 2 is the optimal value to calculate *CF*s radius, trade-off between detecting new clusters without creating outliers.

- *Clustree*: *horizon* = 1000, *maxHeight* = 14.

We give the tree much larger capacity as compared to default parametrization. According to the authors in [KABS11], height of 7 to 11 were compared, with higher heights rendering lower granularities which supposedly let the offline phase achieve higher quality clustering by using conventional stream clustering algorithms. $Height = 14$ increases substantially the detail in granularity as compared to default parametrization and also having in mind the values used in the original paper.

- *Denstream (with DBSCAN)*:

When trying to use the parametrization specified by the authors in [CEQZ06], we realize that the value ranges are not allowed by MOA. We can therefore not replicate exactly its values. We have tried to fine-tune the parameters according to the paper and according to the test scenarios, but the ranges were again not accepted by MOA. A reason for this could be the normalized environment where MOA works, so we tried to scale-down all parameters accordingly but still, proportions could not be maintained for all parameters. Since *Denstream* is the only algorithm available that allows arbitrary-shaped clustering, we decide to, at least, use a configuration that works, keeping in mind that it is most likely not an optimal one. We use neighborhood $\epsilon = 0.16$ to define a bigger data neighborhood than what default parametrization uses.

Table 14 shows first the scenarios with $2d$ visual representation of what the tests would look like in MOA. Table 15 shows the quality results of each algorithm in each scenario. The table contains:

10 scenarios * 4 dimensions * 2 (default VS optimal) parametrization = 80 different configurations.

Parametrization for each configuration is different, while respecting the scenario. Each configuration requires execution of each algorithm to gather 7 measures. **80 configurations * 3 runs[75] = 240 tests**. Each test requires the corresponding parametrization for each algorithm and generates 7 quality metrics. Performing 10 runs per test would render 2400 test in total, which is unfeasible[76], so we run them only once.

[75]Each configuration requires three runs, first *StreamLeader* VS *Clustream*, second *StreamLeader* VS *Denstream*, third *StreamLeader* VS *Clustree*.

[76]We will perform 10 runs per tests when we proceed with scalability testing.

		Scenario 1	Scenario 2	Scenario 3	Scenario 4	Scenario 5
		FEW BIG CLUSTERS 10% NOISE	FEW SMALL CLUSTERS 10% NOISE	MANY SMALL CLUSTERS 10% NOISE	MEDIUM AMOUNT SMALL/BIG CLUSTERS 10% NOISE	MANY SMALL/BIG CLUSTERS 10% NOISE
		SLeader | CluSTR | DenSTR | CTree	SLeader | CluSTR | DenSTR | CTree	SLeader | CluSTR | DenSTR | CTree	SLeader | CluSTR | DenSTR | CTree	SLeader | CluSTR | DenSTR | CTree
DIM SPACE d=2, 5, 20, 50	Default Param Optimal Param					

		Scenario 6	Scenario 7	Scenario 8	Scenario 9	Scenario 10
		FEW BIG CLUSTERS 33% NOISE	FEW SMALL CLUSTERS 33% NOISE	MANY SMALL CLUSTERS 33% NOISE	MEDIUM AMOUNT SMALL/BIG CLUSTERS 33% NOISE	MANY SMALL/BIG CLUSTERS 33% NOISE
		SLeader | CluSTR | DenSTR | CTree	SLeader | CluSTR | DenSTR | CTree	SLeader | CluSTR | DenSTR | CTree	SLeader | CluSTR | DenSTR | CTree	SLeader | CluSTR | DenSTR | CTree
DIM SPACE d=2, 5, 20, 50	Default Param Optimal Param					

Table 14: Synthetic quality testing: visual examples of the scenarios

		Scenario 1 - FEW BIG CLUSTERS 10% NOISE				Scenario 2 - FEW SMALL CLUSTERS 10% NOISE				Scenario 3 - MANY SMALL CLUSTERS 10% NOISE				Scenario 4 - MEDIUM AMOUNT SMALL/BIG CLUSTERS 10% NOISE				Scenario 5 - MANY SMALL/BIG CLUSTERS 10% NOISE				Scenario 1..5 - AVG ALL SCENARIOS 10% NOISE			
		SLeader	CluSTR	DenSTR	CTree	SLeader	CluSTR	DenSTR	CTree	SLeader	CluSTR	DenSTR	CTree	SLeader	CluSTR	DenSTR	CTree	SLeader	CluSTR	DenSTR	CTree	SLeader	CluSTR	DenSTR	CTree
DIM SPACE d=2	Default Param	0.62	0.65	0.25	0.65	0.83	0.77	0.73	0.75	0.71	0.79	0.63	0.77	0.74	0.71	0.53	0.74	0.57	0.67	0.32	0.67	0.69	0.72	0.49	0.71
	Optimal Param	0.62	0.65	0.48	0.67	0.83	0.77	0.51	0.77	0.91	0.84	0.23	0.82	0.76	0.71	0.34	0.74	0.67	0.68	0.33	0.71	0.75	0.73	0.38	0.74
DIM SPACE d=5	Default Param	0.80	0.83	0.23	0.80	0.87	0.81	0.70	0.81	0.90	0.84	0.32	0.85	0.92	0.83	0.54	0.83	0.91	0.86	0.31	0.84	0.88	0.83	0.42	0.83
	Optimal Param	0.82	0.83	0.5	0.84	0.87	0.81	0.69	0.82	0.92	0.93	0.28	0.90	0.92	0.83	0.55	0.87	0.92	0.90	0.28	0.90	0.89	0.86	0.46	0.87
DIM SPACE d=20	Default Param	NA	NA	NA	NA	0.90	0.74	0.71	0.80	0.95	0.69	0.37	0.57	0.93	0.81	0.70	0.85	0.95	0.64	0.32	crash	0.93	0.72	0.52	0.68
	Optimal Param	NA	NA	NA	NA	0.91	0.74	0.72	0.80	0.96	0.91	0.37	0.72	0.93	0.81	0.72	0.87	0.95	0.92	0.35	0.75	0.94	0.84	0.54	0.78
DIM SPACE d=50	Default Param	NA	NA	NA	NA	0.91	0.55	0.66	0.63	0.95	0.24	0.32	crash	0.93	0.55	0.55	0.77	0.95	0.42	0.36	0.69	0.93	0.44	0.47	0.64
	Optimal Param	NA	NA	NA	NA	0.91	0.55	0.67	0.66	0.96	0.67	0.32	0.42	0.93	0.73	0.58	0.70	0.95	0.73	0.36	0.64	0.93	0.67	0.48	0.60

		Scenario 6 - FEW BIG CLUSTERS 33% NOISE				Scenario 7 - FEW SMALL CLUSTERS 33% NOISE				Scenario 8 - MANY SMALL CLUSTERS 33% NOISE				Scenario 9 - MEDIUM AMOUNT SMALL/BIG CLUSTERS 33% NOISE				Scenario 10 - MANY SMALL/BIG CLUSTERS 33% NOISE				Scenario 6..10 - AVG ALL SCENARIOS 33% NOISE			
		SLeader	CluSTR	DenSTR	CTree	SLeader	CluSTR	DenSTR	CTree	SLeader	CluSTR	DenSTR	CTree	SLeader	CluSTR	DenSTR	CTree	SLeader	CluSTR	DenSTR	CTree	SLeader	CluSTR	DenSTR	CTree
DIM SPACE d=2	Default Param	0.57	0.57	0.21	0.57	0.86	0.59	0.54	0.60	0.68	0.64	0.51	0.64	0.73	0.61	0.49	0.62	0.57	0.59	0.29	0.60	0.68	0.60	0.40	0.60
	Optimal Param	0.57	0.57	0.46	0.58	0.88	0.59	0.39	0.61	0.89	0.63	0.28	0.68	0.8	0.61	0.34	0.67	0.67	0.60	0.35	0.63	0.76	0.60	0.36	0.63
DIM SPACE d=5	Default Param	0.69	0.59	0.28	0.61	0.80	0.70	0.44	0.69	0.93	0.80	0.35	0.85	0.89	0.81	0.36	0.84	0.87	0.74	0.33	0.82	0.84	0.73	0.35	0.76
	Optimal Param	0.69	0.59	0.36	0.63	0.83	0.70	0.59	0.71	0.95	0.85	0.48	0.89	0.89	0.81	0.52	0.85	0.87	0.83	0.34	0.91	0.85	0.76	0.46	0.80
DIM SPACE d=20	Default Param	NA	NA	NA	NA	0.88	0.51	0.43	0.78	0.95	0.34	0.34	0.85	0.89	0.51	0.49	0.77	0.93	0.33	0.33	0.80	0.90	0.42	0.40	0.80
	Optimal Param	NA	NA	NA	NA	0.91	0.51	0.45	0.78	0.96	0.66	0.35	0.86	0.89	0.51	0.57	0.76	0.94	0.84	0.35	0.84	0.92	0.63	0.43	0.81
DIM SPACE d=50	Default Param	NA	NA	NA	NA	0.87	0.47	0.35	0.59	0.90	0.12	0.31	crash	0.92	0.39	0.38	crash	0.92	0.12	0.32	crash	0.90	0.27	0.34	0.52
	Optimal Param	NA	NA	NA	NA	0.90	0.47	0.40	0.54	0.94	0.34	0.32	0.40	0.92	0.42	0.37	0.69	0.93	0.43	0.32	0.40	0.92	0.41	0.35	0.51
		0.67	0.66	0.35	0.67	0.87	0.64	0.56	0.71	0.90	0.64	0.36	0.70	0.87	0.66	0.50	0.75	0.85	0.64	0.33	0.70				

Table 15: Synthetic quality testing: results for each scenario

The table shows the throughout testing of each algorithm on each scenario. On the right side, we can also see average results per algorithm for all scenarios and split per 10% and 30% noise respectively. Below, another summary per scenario type with no split by noise levels. On both summaries, the best result of the four algorithms is underscored.

From each test, for each algorithm, we take the resulting 7 quality metrics mentioned before (*CMM*, *Rand Statistic*, *Silhouette Coefficient*, *Homogeneity*, *Completeness*, *F1-P*, *F1-R*) and calculate the final overall quality as $Q_AVG = \frac{CMM+RI+SC+h+c+F1_P+F1_R}{7}$.

First thing we notice from the tests, is that *Scenario 1* and *Scenario 6* have *NA*[77] results for dimensionality 20 and 50. The reason is that MOA can not create *big* (i.e radius 0.11 to 0.25) clusters with such dimensionality, prompting to reduce radius in cluster creation and/or split. In general, working in medium (20) to high dimensionality(50), we had to reduce the cluster size in order to run the tests. Looking at the numeric results, MOA's visual representation of the clusters and evolution of the metrics as the test proceed, we can see multiple situations were the algorithms react interestingly. We can not display each situation and scenario graphically, so we draw some general conclusions looking at the numbers, and also describing how the algorithms handle the scenarios with some visual representations:

1) *Denstream* is the worst performer most of the times:
as mentioned before, we could not emulate the author's suggested parametrization in MOA for the optimal setting, therefore we had to choose one that at least executed. MOA's default setting was also different from author's, so we should look at *Denstream*'s results with caution.

2) *Clustree* crashed in five scenarios:
this happened mostly in high dimensionality and with default parametrization. It is out of scope to investigate the reasons of the crash so we will no comment on this further. In order to produce the statistics, a value of 0.5 has been assigned to situations were the algorithm crashed. This value seems reasonable since sometimes it could be a good result in a difficult scenario but sometimes could be low when other algorithms ave good performance.

3) All algorithms have problems with heavy cluster overlapping in very low dimensional spaces ($2d$):
scenario 1 and 6 (FEW BIG CLUSTERS) seem to present challenging conditions for all four algorithms. This is because when we have big clusters changing position, the lower the dimensionality the higher the chances of overlapping and collisions. This has the effect that points will be found in the same location in d-dimensional space, therefore proximity metrics return a small distances and high similarity. The effect in *StreamLeader*, for instance, is that when two clusters get close enough, merging operations are triggered (Property 2 *LeaderKernel* additivity and *merge* operation), returning one cluster instead of the several that might exist as ground-truth. So when collisions do happen, StreamLeader tends to merge. All algorithms suffer from the same effect, although they have an (unfair) important advantage because *Clustream* and *Clustree* use k-means as offline conventional algorithm, receiving the true number of clusters from MOA, so the chances to deliver better clustering in these situations are higher.
We can see in Figure 37 a simplification of this effect caused by two clusters represented by two *LeaderKernels* (contour in red). The two clusters are identified in space in the upper side. In the middle, when they get close enough, *StreamLeader* detects the instances and tries to encapsulate them all with a unique leader by calculating the center of gravity according to $DIST_{SL}$ function. Lastly, when the clusters overlap totally, all algorithms tend to represent the two ground-truth clusters with a unique one. We observe quality Metrics dropping down in general in such situations, specifically *homogeneity*, *F1-P* and *F1-R* which reflect the errors properly. The way to properly parametrize *StreamLeader* in these situations, is with the assignment of a smaller (than biggest cluster radius) D_MAX. In this way, it tries to identify the masses first and then expand from inside out (expansion capabilities), as opposite to contract from outside down (contraction capabilities).

[77] *NA* = Not Available.

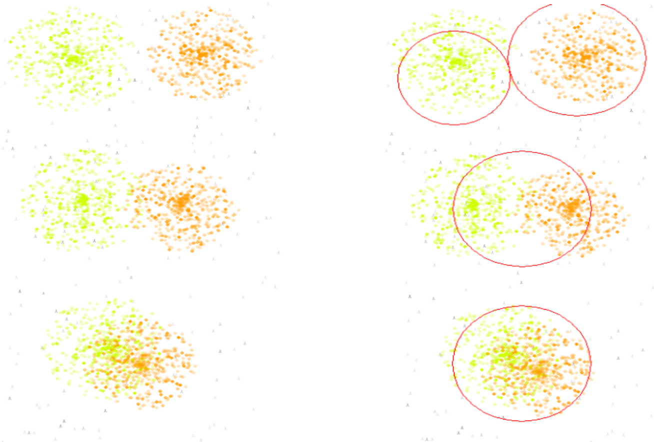


Figure 37: Two *LeaderKernels* mapping true clusters (top) merge if true clusters get close enough (bottom)

4) Dimensionality higher than $2d$:
StreamLeader handles well situations with higher (greater than 2) dimensionality because of its design. Collisions are not so often, sparsity of the data is somehow expected, but inter-cluster distance is normally higher than inter-cluster distance among instances. *StreamLeader* captures the masses first, and can expand if necessary to a bigger cluster (default parametrization). With optimal D_MAX it normally does not need to expand, and because collisions are not so frequent, it can contract and look for the most appropriate radius to capture the mass detected. *Clustream*, however, shows a noticeable decrease in quality results when dimensionality increases. *Clustree* also, but not the same extend. The reason is caused by the use of cluster feature vectors *CFs* together with a conventional clustering algorithm. In high dimensions, *CFs* do not have the capabilities to contract (like *LeaderKernels* do), therefore *CFs* tend to become bigger, occupying more space and distorting their centers. Since *CFs* are taken as pseudo-points and the centers are indeed distorted, then the clustering delivered is normally over-sized and returned clusters overlap with one another in space. *Clustream* can attenuate this negative effect when number of ground-truth clusters increases by augmenting the *micro-ratio* to a factors bigger than 10. When this happens, it starts mapping points more closely, so the *CFs* tend to be of smaller radius. This has the effect of increasing the resulting clustering quality by taking more *CFs*, each representing a smaller and smaller set of points. But also two considerable drawbacks: the performance gets degraded considerably (as we will see in scalability testing), and second such low level mapping produces some sort of *overfitting*, by mapping also the noise. In high noise scenarios, quality drops substantially.
We can se the effect of high dimensionality with default parametrization on the algorithms in Figure 38. Upper left plot shows true clusters (or ground-truth) in high d with 10% noise. Upper right shows the clustering provided by *StreamLeader*, where the *LeaderKernels* adjust remarkably well to the true clusters. Middle picture shows *Clustream* producing over-sized and overlapping micro-clusters (in green) and clustering (in blue). Bottom left shows *Clustree* with similar problems and bottom right *Denstream*, where it maps almost exactly the points with many clusters.

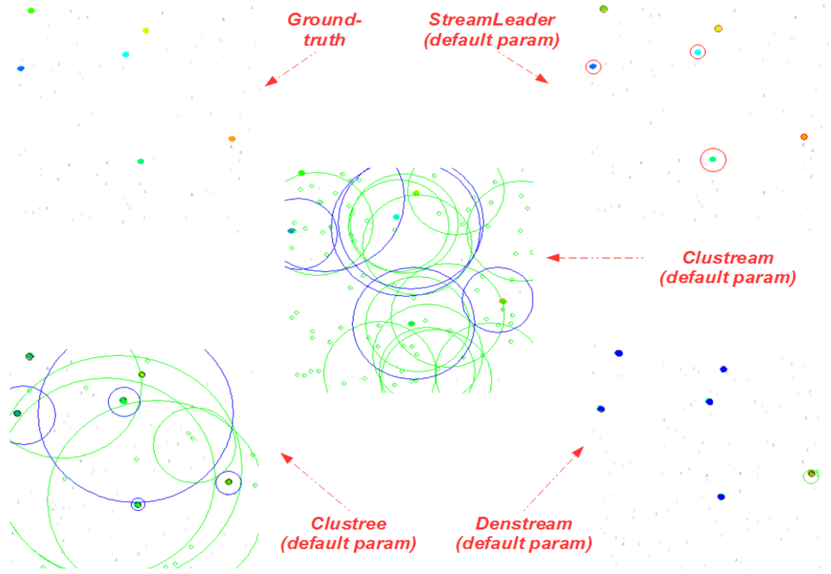


Figure 38: micro-clusters (green) and clustering (blue) suffer distortions in high d in *Clustream*, *Clustree* using default parametrization

Same display with optimal parametrization in Figure 39:

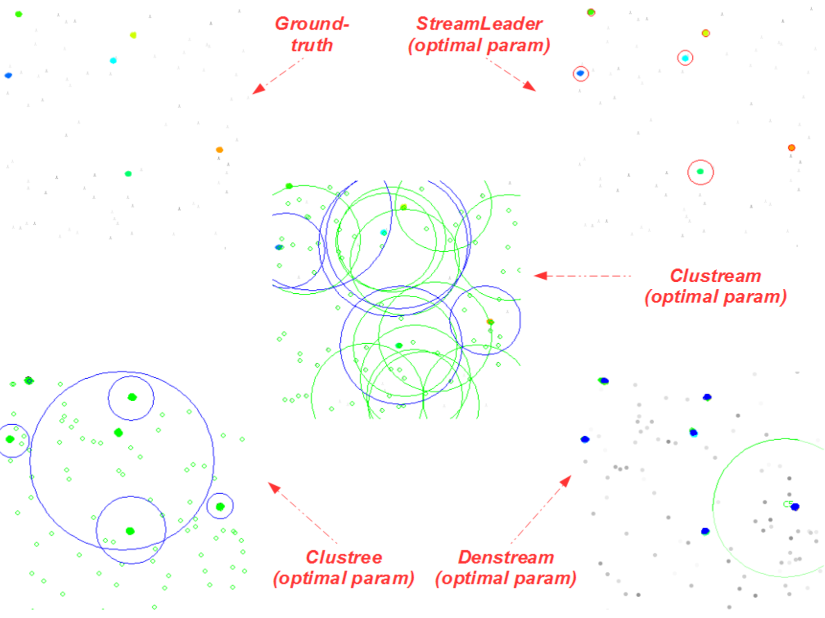


Figure 39: micro-clusters (green) and clustering (blue) suffer distortions in high d in *Clustream*, *Clustree* using optimal parametrization

We observe that *StreamLeader* shows no difference in clustering delivered. This is because contraction capabilities work well with default D_MAX. *Clustream*, in this case behaves the same, because *micro-ratio* was already above 10 with 7 clusters. In case we had a situation with 30 clusters, we would have observed that optimal parametrization would have required around 300 *CF*'s, and we would observe how each mapped closely to each instance, producing the mentioned *overfitting-like* effect. Still it shows how big and small *CF*'s overlap in space, covering in many situations one another and delivering the offline phase a distorted view of the instances. *Clustree* shows how increasing height of tree from default 8 to 14, *overfitting-like* also occurs. Delivered clustering produces a really big cluster, covering most of the d normalized space, and overlapping with other clusters. We recall that these representation in $2d$ does not imply overlapping of clustering, but in this case, other views on other dimensions would show it. Also, the quality metrics reflect this overlapping in clustering with a drop.

5) Noise:
it poses a significant challenge for the algorithms except *StreamLeader*, which only experiments small de-

creases in quality. We can see in Figure 40, in the upper left side, seven clusters of different sizes in a stream with 33% noise in high *d*.

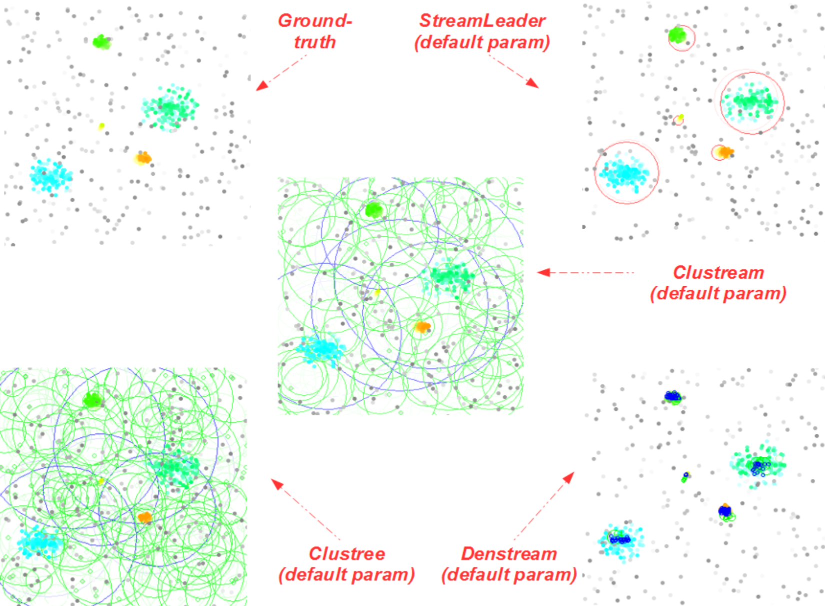


Figure 40: micro-clusters (green) and clustering (blue) suffer distortions with heavy noise in *Clustream*, *Clustree*

In the upper right side, *StreamLeader* performs remarkably well, detecting the true clusters and building *LeaderKernels* describing them, using contraction capabilities. But the key in these scenarios is the noise treatment it carries out in two phases, first percentile cuts and logarithmic cuts subsequently. The two techniques combined seem to be very effective eliminating the heavy noise. In very heavy noise scenarios we observed sometimes the effect of a *LeaderKernel* capturing noise and surviving the noise treatment, but it dissipated very quickly, distorting the quality measures only momentarily. *Clustream* and *Clustree*, in the middle and bottom left corner respectively, have considerable problems because their *CF*'s absorb the noise as if they were points, overlapping heavily with one another. Furthermore, this distortion is passed on to the offline clustering algorithm producing over-sized (they include the noise) and overlapping clustering, causing drops in quality measures. *Denstream* in the bottom right side, seems to detect the very center of mass of the cluster, although not covering the required space. MOA metrics still deliver bad quality results, so we will not comment further due to the problem with parametrization.

6) Optimal parametrization outperforms default parametrization:
this is the case for all algorithms, but there are important differences. *StreamLeader* improves quality only marginally on each scenario. This is an indication that *LeaderKernels* expand and contract properly, adjusting their radius to the data and away from D_MAX if needed. Only in one test type we can see substantial improvement, *Scenario* 3 & 8 with 2 dimensions and many small clusters, represented in Figure 41. The reason is that those small clusters are found in very small space, so several can fall under the attraction of D_MAX and therefore placed in the same *LeaderKernel*. Optimal setting means decreasing D_MAX to a value similar of the size of radius of the true clusters. In this way, it does not capture several clusters and concentrates only on the small mass represented by one single cluster. A mixture of small and big clusters in very low *d* also challenges default parametrization. Still, this effect disappears when we move into 5, 20, 50 dimensions. We can see this effect in the picture below:

We already notice that optimal parametrization has no noticeable performance effects on *StreamLeader*, although this will be properly checked in scalability testing in next section.

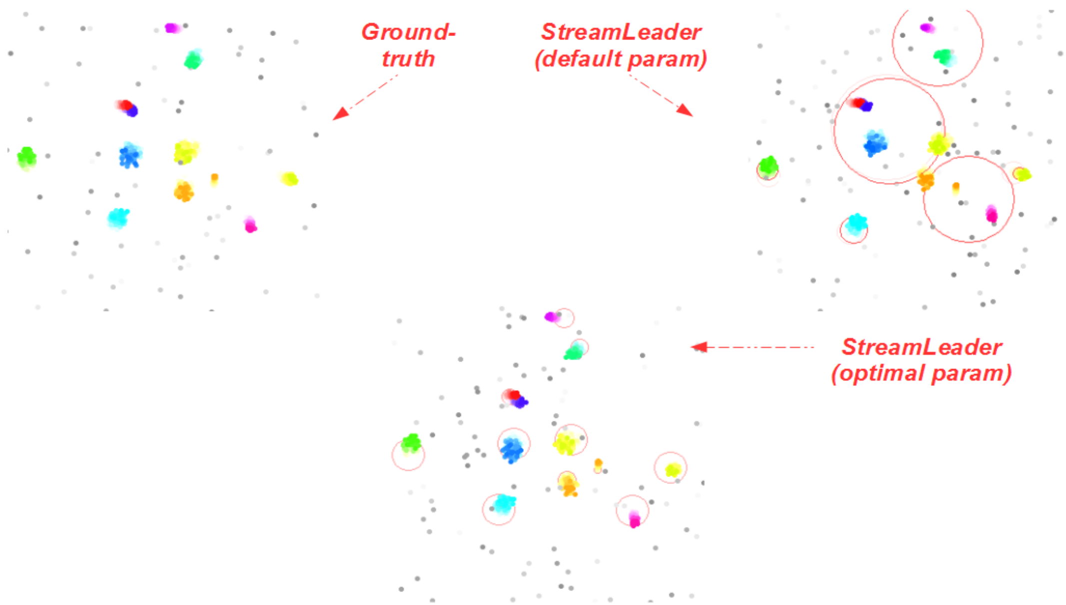


Figure 41: *StreamLeader* delivering clustering (in red) with default & optimal parametrization in $2d$ space

With regards to *Clustream*, optimal parametrization depends on keeping *micro-ratio* above 10. This happens mostly in scenarios 3 & 5, with numerous clusters (20 or more). More micro-clusters are therefore needed (above 200 for a 20 cluster scenario, or above 300 with 30) which will then describe the stream with a finer granularity, lowering substantially the radius of the *CF*s micro-clusters. This has the effect of mapping also noise (and noticeable negative effects on performance, amplified when we move into higher dimensionality). Figure 42 shows the effects described above, with 24 true clusters as ground-truth in the plot in the upper side, micro-clusters (in green) and clustering (in blue) delivered by *Clustream*, with default parametrization on the left side and optimal on the right side.

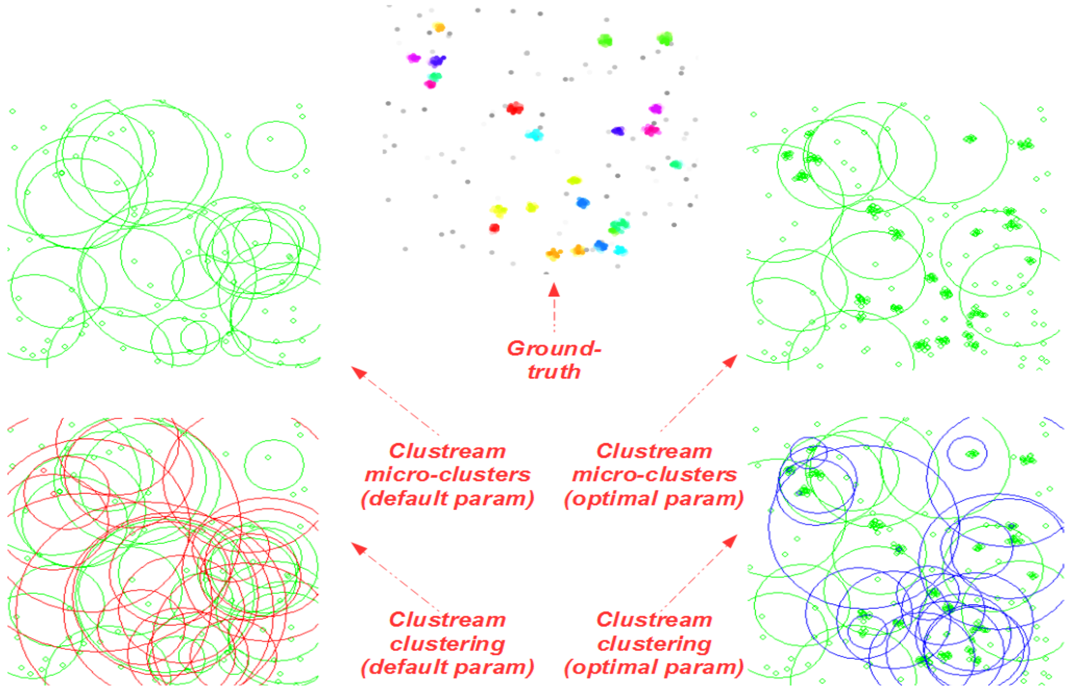


Figure 42: *Clustream* clustering with default & optimal parametrization (micro-clusters green, clustering red & blue)

We can observe that with default parametrization (100 micro-clusters), *micro-ratio* is below factor 10 ($\frac{100}{24} = 4.16 \leq 10$) as recommended by the authors. Micro-clusters are therefore bigger and offline clustering produces heavily over-sized and overlapping clusters (bottom-left corner in the picture above), causing degradation in clustering quality metrics. On the other hand, optimal parametrization implies manual

increase of number of micro-clusters to 260 to keep *micro-ratio* above 10 ($\frac{260}{24} = 10.8 \geq 10$). This produces the effect of decreasing micro-cluster's size and also mapping data with lower granularity, which produces an *overfitting-like* effect. This can be seen in the picture on the middle-right side, as very small micro-clusters group together and describe almost totally the cluster, together with bigger ones occupying big portions of the 20d space. Clustering produced is of higher quality than that of default parametrization, although high dimensionality causes the offline algorithm to still produce overlapping due to the micro-clusters. We should also note that this effect would be amplified with high noise levels producing further drops in quality. We also experience substantial negative impact on runtime performance (further details on this effect on the scalability section).

Clustree suffers from same effects. Optimal parametrization of 14 levels of the tree provides finer granularity *CF*s than height 8 to summarize the data. It can increase the quality, but it is also very sensitive to noise. In Figure 43, we can see same stream as shown in 42, 24 true clusters in the upper side, the effects of both parametrizations for the *Clustree*, default in the left, optimal in the right, and comparison with clustering delivered by *StreamLeader* at the bottom, all in same scenario at same instant. *StreamLeader* achieves higher accuracy, contracting in d space where the mass is and producing no overlapping. *Clustree* suffers from overlapping and oversize clustering, regardless of parametrization used.

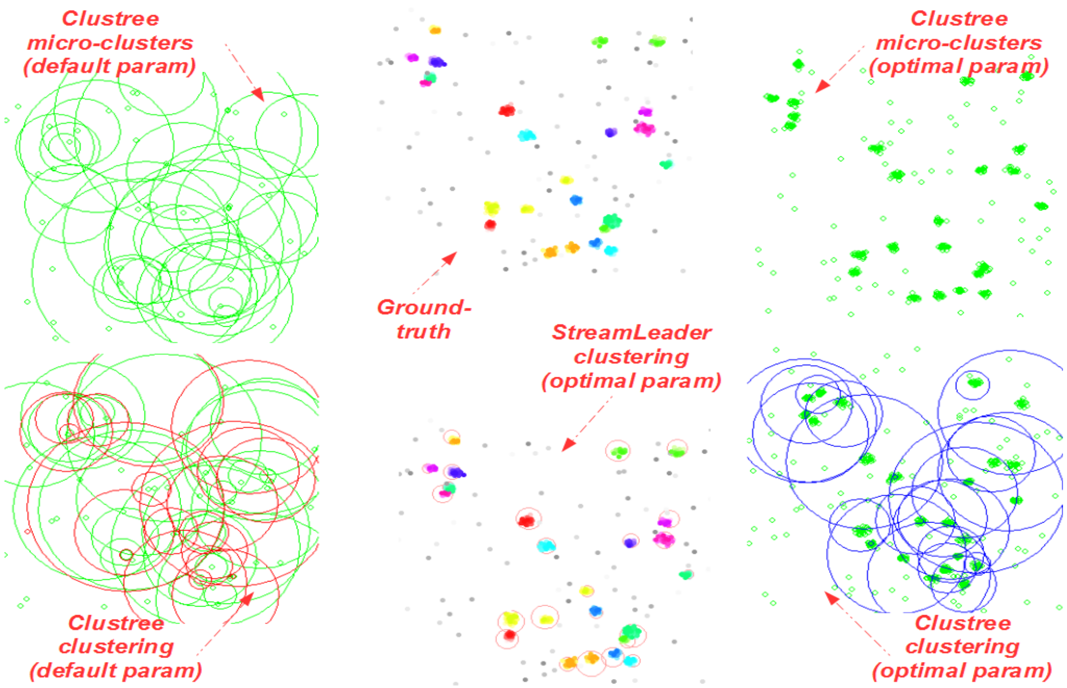


Figure 43: *Clustree* clustering with default & optimal parametrization. Also *StreamLeader* with optimal

7) Looking at quality results, *StreamLeader* outperforms, in average, *Clustream*, *Denstream*, *Clustree*: results based on dimensionality and on noise levels indicate that *StreamLeader* delivered better quality clustering in average, both in default and optimal configurations. Only in dimensionality 2 with default parametrization, *StreamLeader* was outperformed by *Clustream* due to the high collision of clusters and the advantage that *Clustream* has of using k-means with the real number of true clusters provided by MOA. Per scenario type, it outperforms in average again the contenders, only sharing the highest score with *Clustree* in scenario with few big clusters combining 10% & 33% noise levels.

8) Individual clustering quality metrics: default vs optimal parametrization:
for each of the 216 tests[78], we take the 7 quality metrics used. Figures 44, 45, 46 and 47 show how each algorithm performs on each metric in average, with default parametrization on the left side and optimal in the right. We want to see: a) quality achieved and b) effect of parametrization on each metric:

[78]Experiments consisted of 240 tests except 16 that crashed in MOA, from scenario 1 & 6 (few big clusters with 10% & 33% noise levels) in dimensionality 20 & 50.

Figure 44: Synthetic results: average *CMM* and *Silhouette Coef*, default vs optimal parametrization

Figure 45: Synthetic results: average *F1-P* and *F1-R*, default vs optimal parametrization

Figure 46: Synthetic results: average *Homogeneity* and *Completeness*, default vs optimal parametrization

Figure 47: Synthetic results: average *Rand Statistic* and overall Q_AVG, default vs optimal parametrization

With regards to parametrization, we can extract important information from the figures:

- with default parametrization, *StreamLeader* seems to outperform all other algorithms in each of the metrics with exception of *Completeness*. However, this was somehow anticipated because, as we mentioned before, techniques like *Clustream* filled the d space with over-sized *CF*s and overlapping clusters due to the use of conventional clustering algorithms. When that is the case, *Completeness* renders very high quality result because all members of a single class are always assigned to the same cluster. But this also has a very negative collateral effect on *homogeneity*, which delivers quality when each cluster contains members of a single class, which in this case hardly happens because big overlapping clusters contain much more than just one class. This is also noticeable in *F1-R* recall which drops in quality when clusters are over-sized including instances of one class but also from other classes or just noise. *F1-P* shows higher values since bigger clusters would retrieve most of the instances of a class. Last graph shows the average of all quality measures, showing *StreamLeader* performing remarkably well and *Clustree* outperforming clearly *Clustream*

- with optimal parametrization, *StreamLeader* seems to outperform all other algorithms in each of the metrics again with exception of *Completeness*. The reasons are as above. We notice also that *StreamLeader* only improves marginally (default scored high already). This indicates that the contraction and expansion capabilities of the *LeaderKernels* work well and capture the mass of the clusters effectively. Another take from the plots is that the other algorithms do improve substantially their quality, specially *Clustream* in *F1-P*, *F1-R*, *homogeneity* and *Rand Statistic*, which indicates that the *micro-ratio* needs to be adjusted properly with the number of real clusters. *Clustree* improves also but to a lesser extend, still, showing a better overall performance than *Clustream*. *Denstream* showed poor performance, probably to the inadequate parametrization.

9) Average clustering quality metrics: 10% vs 33% noise levels:
we do the same exercise averaging all metrics for all tests per noise level and produce the results shown in Figure 48 in the upper plots. For 10% noise environments in the left, *StreamLeader* seems to outperform the other algorithms, while *Clustream* and *Clustree* have a similar performance. Introducing 33% noise levels, in the right, does not seem to impact substantially *StreamLeader*, which indicates that the algorithm has high tolerance to noise thanks to the combination of percentile, logarithmic cuts and the avoidance of conventional clustering algorithms. *Clustree* gets impacted but not as much as *Clustream*, which experiences a substantial drop in quality. This seems to corroborate what we expected since offline phase takes as input the noise captured by micro-clusters, which produced degradation in clustering.

10) Average clustering quality metrics: low dimensionality (d = 2, 5) vs medium/high (d = 20, 50):
lastly, we analyze the general performance focusing on dimensionality. In Figure 48 in the lower plots, we see that, in low dimensional space ($d = 2, 5$), *StreamLeader* outperforms again *Clustream* and *Clustree*, being these two similar in performance. *Denstream* lags behind. When we move into medium/high dimensionality ($d = 20, 50$), in the bottom right, *StreamLeader* delivers even higher quality. Reason, among others, is that no offline conventional clustering algorithm is used based on *LeaderKernels*, which would tend otherwise to create over-sized clusters. Also, collisions in high d are normally reduced, allowing *StreamLeader* to focus only on detecting the concentrated mass using expansion/contraction capabilities. Effective noise treatment also contributes to eliminate undesired *LeaderKernels* that captured noise. On the other hand, *Clustree* experiences substantial degradation in performance, although not as much as *Clustream*, which is heavily affected by noise. One of the reasons for their lower performance is that they collect as many *CF*s as they can maintain in order to capture fine-grain encapsulation of the data stream. In higher dimensional space, hyper-spherical micro-clusters tend to grow bigger and contain everything they find in d space, including noise. Passing these encapsulations to the offline phase where conventional algorithm takes them as input (which can be already distorted) might cause degradation in clustering, including over-sized and overlapping clustering. If they use optimal parametrization to maintain a larger number of CFs, then their radius is reduced, they summarize in a very low detail, producing what it looks as *overfitting-like* capturing then everything, including noise with individual *CF*s

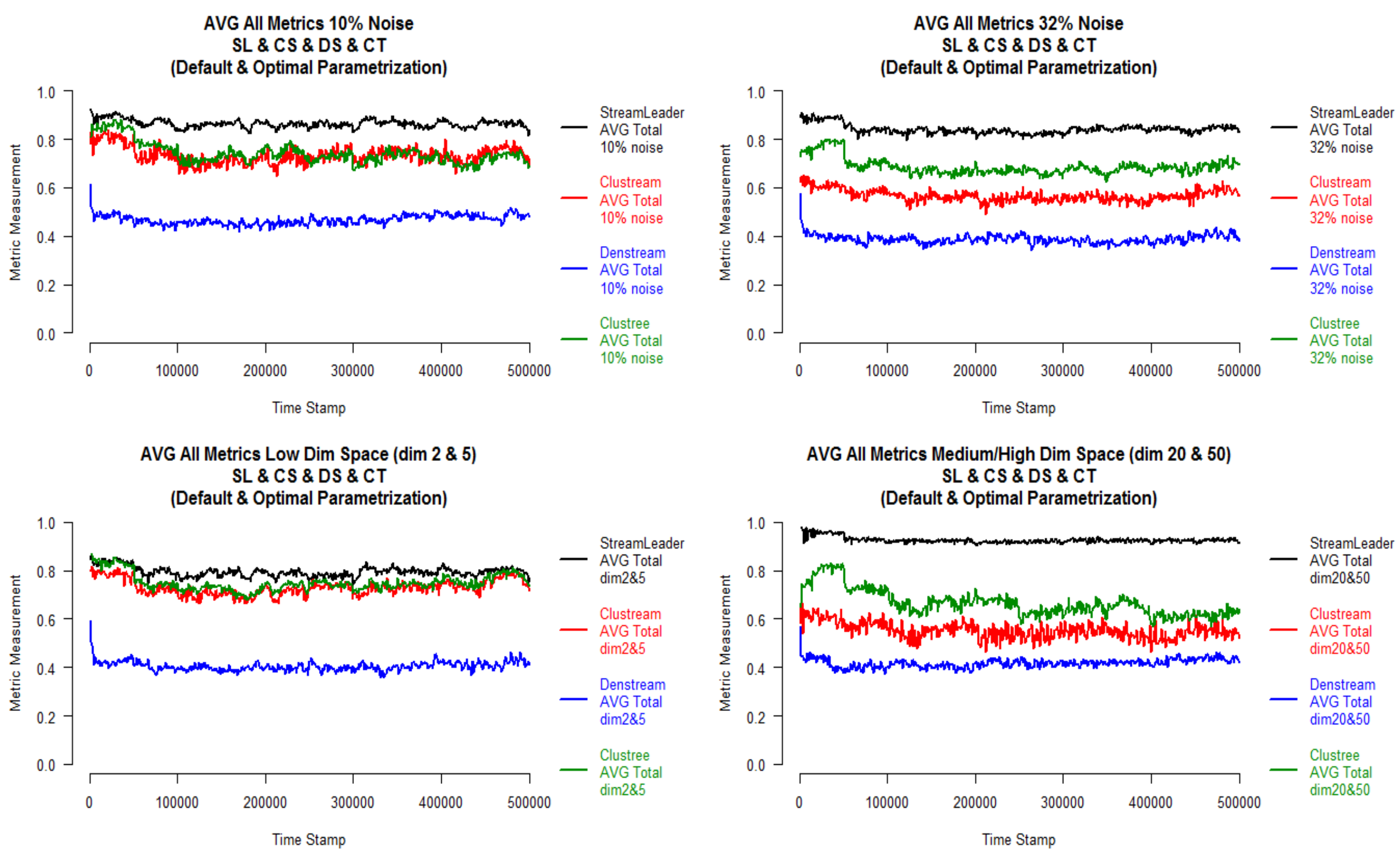


Figure 48: Synthetic results: average overall quality Q_AVG per noise levels and dimensionality

11) Overall Average clustering quality metrics:
Finally, we average all metrics for all synthetic tests ($Q_AVG = \frac{CMM+RI+SC+h+c+F1_P+F1_R}{7}$) per algorithm and produce the following the plot shown in Figure 49 which describes the overall performance with synthetic data. *StreamLeader* outperforms in quality *Clustream*, *Clustree* and *Denstream*. This confirms that *LeaderKernels* describe efficiently the synthetic data stream, time window management handles well their aging, noise treatment is effective eliminating disturbances in underlying data generation and the attempt of using no conventional clustering worked remarkably well:

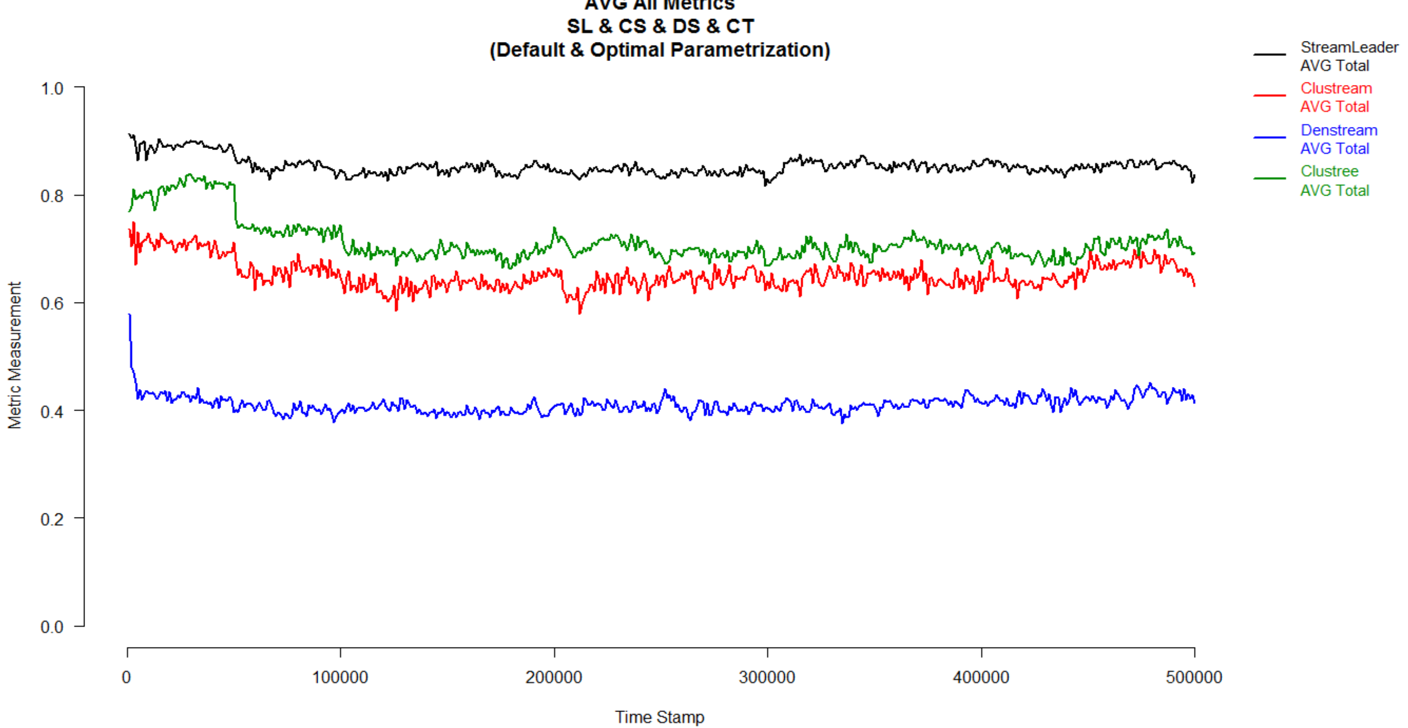


Figure 49: Synthetic results: overall quality performance for all scenarios

3.4 Quality tests - Real Data

Synthetic data is very useful when it comes to designing scenarios with different configurations, coming normally in the form of Gaussian distributed points around centers in $\mathbb{R}^d$. Real data sets can behave differently to those artificially generated though. Thus, we need to find real data that evolves significantly. Reviewing the literature, we find two datasets that have been used in clustering tasks and later in stream clustering:

- Forest Cover Type (KDD)
- Network Intrusion Detection (KDD-CUP'99)

MOA provides the capability to feed data files by using the class *FileStream*. By adding headers to the file, which explain the content of the data, MOA converts all observations contained into a data stream that can be fed to the algorithms. An example for such headings is the following:

```
@RELATION Forest_Cover_Type_MOA_input
@ATTRIBUTE Elevation NUMERIC
@ATTRIBUTE Aspect NUMERIC
...
...
@ATTRIBUTE Cover_Type_desc {Spruce_Fir,Lodgepole_Pine,Ponderosa_Pine,Cottonwood_Willow,Aspen,Douglas_fir, Krummholz}

@DATA
0.366561630964665,0.157946610042229,0.0310230063453802,0.152063116587451,0.21612095140829,0.0553723245473644, ... ,Aspen
...
```

3.4.1 Data Set 1: Forest Cover Type

Forest Data Cover data set has been used frequently in the stream clustering literature, in *StreamKM++* [AMR+12] or *Clustree* [KABS11] among others. We will use it then in MOA to compare the algorithms. Table 16 contains a brief description of the dataset:

Real Data Set Name	**Forest Cover Type (KDD)**
Size	580K observations
Goal	Cluster the 7 different types of forest trees found in the terrain
Description	The forest cover type for 30 x 30 meter cells obtained from US Forest Service (USFS) Region 2 Resource Information System (RIS) data. It contains 10 continuous variables: (Elevation, Aspect, Slope, Horizontal_Distance_To_Hydrology, Vertical_Distance_To_Hydrology Horizontal_Distance_To_Roadways, Hillshade_9am , Hillshade_Noon , Hillshade_3pm , Horizontal_Distance_To_Fire_Points) 7 classes to cluster (forest cover types), in variable Cover_Type_desc: - Spruce/Fir - Lodgepole Pine - Ponderosa Pine - Cottonwood/Willow - Aspen - Douglas-fir - Krummholz
Min classes[79]	2
Max classes[80]	7
Avg classes[81]	4.02
Location	`http://kdd.ics.uci.edu/databases/covertype/covertype.html`

Table 16: *Forest Covert Type* data set details

[79] Minimum number of different classes appearing simultaneously.

We select settings for the data stream (default values $decayHorizon = 1000$, $decayThreshold = 0.01$, $evaluationFrequency = 1000$) and normalize the data. Then execute the streaming of the data and notice again is that visualization in streaming scenarios is an important field to be improved upon. Thankfully, MOA provides some basic visualization in $2d$ modes, which is helpful, but surely techniques like *parallel bars* in *streaming version* would be very useful for this sort of investigation. We even considered developing a technique for this, but we realized that it would be far too ambitions for the scope of this work. We finally select some sets of informative dimensions and see that the classes are intertwined and broken in different pieces, which poses certainly a real challenge to the algorithms. Figure 50 shows four different 2-dimensional screen-shots in MOA of the execution of the streaming of the data set, at one specific instant. This gives an idea of what the data looks like. Upper left picture displays the attributes *Elevation* vs *Aspect*. Upper right *Elevation* vs *Horizontal distance to hydrology*. Bottom left *Elevation* vs *Vertical distance to hydrology* and finally bottom right *Elevation* vs *Horizontal distance to roadways*. We also identify four classes (four types of forest cover) present in the stream at that instant (class 1...class 4). On the right plots, We observe how class 1, class 2 and class 3 appear in the visualization in different places. That already indicates that creating unique non-overlapping clusters for each class will be difficult.

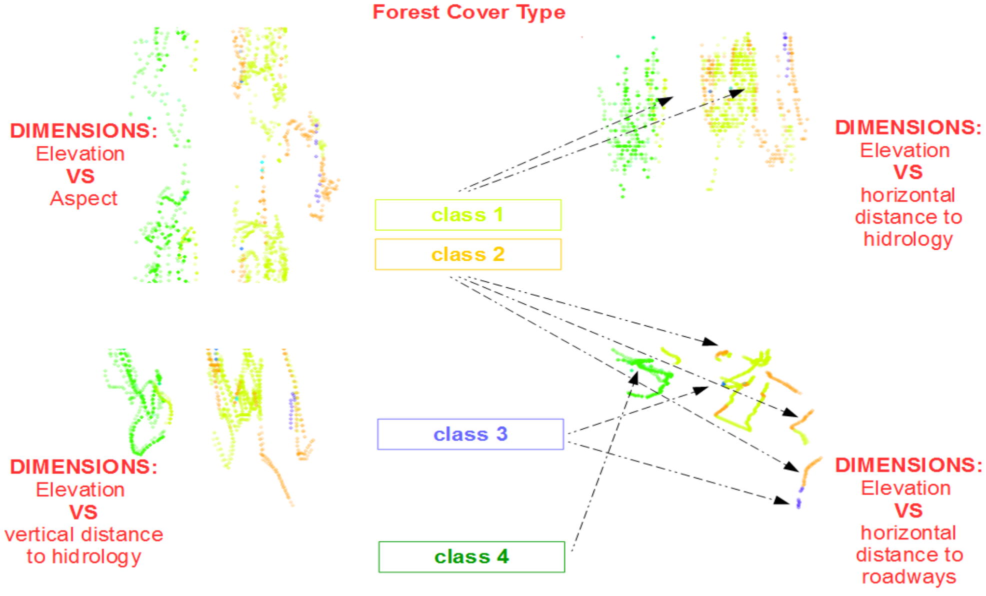


Figure 50: *Forest Cover Type*: visualizing the stream using four different sets of attributes

We also realize that the calculation of the ground-truth or true clustering done by MOA will not be straightforward as seen in Figure 51. MOA does this probably *on-the-fly* by capturing each class (all instances belonging to the same label) with the smallest possible hyper-spherical cluster. We should bear in mind that this step is necessary in order to calculate external clustering measures. We observe seven oversized and overlapping clusters (contour in black) covering each class label (forest cover type, i.e. Lodgepole_Pine, Ponderosa_Pine, etc). Quality measures could well be affected because of this.

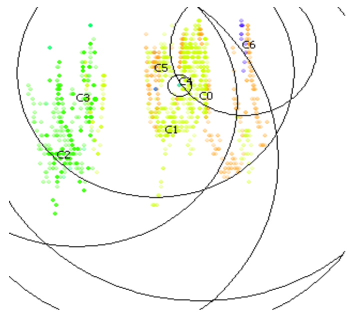


Figure 51: *Forest Cover Type*: true clusters or ground-truth calculated *on-the-fly* by MOA

[80]Maximum number of different classes appearing simultaneously.
[81]Average number of different classes appearing simultaneously.

Table 17 contains the parametrization chosen for the algorithms ($horizon = 1000$ for all):

Algorithm	*StreamLeader*
Default Param	$D_MAX = 0.11$
Optimal Param	$D_MAX = 0.17$

Algorithm	*Clustream*
Default Param	- $MaxNumKernels = 100$ - $kernelRadiFactor = 2$
Optimal Param	- $MaxNumKernels$[82]$= 150$ - kernelRadiFactor=2

Algorithm	*Clustree*
Default Param	$maxHeight = 8$
Optimal Param	$maxHeight = 14$

Algorithm	*Denstream*[83]
Default Param	- $epsilon = 0.02$ - $beta = 0.2$ - $mu = 1$ - $initPoints = 1000$ - $offline = 2$ - $lambda = 0.25$ - $processingSpeed = 100$
Optimal Param	- $epsilon = 0.16$ - $beta = 0.2$ - $mu = 1$ - $initPoints = 1000$ - $offline = 2$ - $lambda = 0.25$ - $processingSpeed = 100$

Table 17: *Forest Covert Type*: parametrization used for *StreamLeader*, *Clustream*, *Denstream* and *Clustree*.

Figure 52 displays the clustering provided by *StreamLeader* (contour in red) and *Clustream* (contour in blue) for the instant shown in Figure 50. Attributes chosen for visualization are *Elevation* vs *Horizontal distance to hydrology*. *Clustream* produces as many clusters (seven) as ground-truth shows, probably because it receives from MOA the true number of clusters to discover, so this help is not to be expected in a real scenario. Still, they do not seem very consistent with the data. *StreamLeader* delivers four in optimal parametrization.

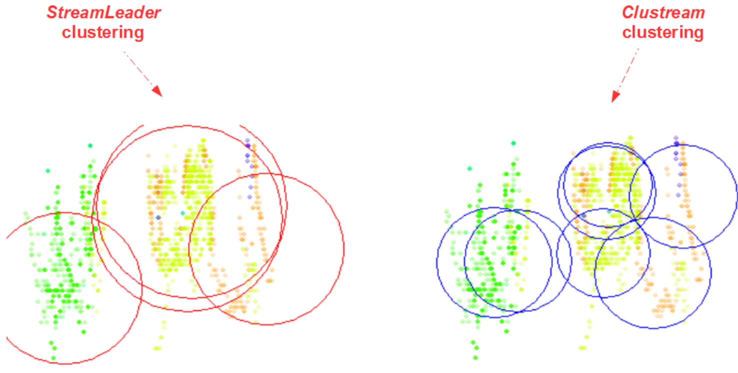


Figure 52: *Forest Cover Type*: clustering by *StreamLeader* (red) and *Clustream* (blue)

[82]While 100 micro-clusters achieves *micro-ratio* ≥ 10, we still opt to increase that ratio to achieve finer granularity.
[83]We recall that we could not replicate the author's preferred parametrization for *Denstream* in MOA.

Finally we gather quality metrics for both default and optimal parametrization in Figures 53 (*CMM* and *Silhouette Coef*), 54 (*F1-P* and *F1-R*), 55 (*Homogeneity* and *Completeness*), 56 (*Rand Stat* and *Q_AVG*):

Figure 53: Forest Cover Type: *CMM* and *Silhouette Coef* default vs optimal parametrization

Figure 54: *Forest Cover Type*: *F1-P* and *F1-R* on default vs optimal parametrization

Figure 55: *Forest Cover Type*: *Homogeneity* and *Completeness* on default vs optimal parametrization

Figure 56: *Forest Cover Type*: *Rand Statistic* and Q_AVG on default vs optimal parametrization

Together with the overall performance with both parametrizations in Figure 57:

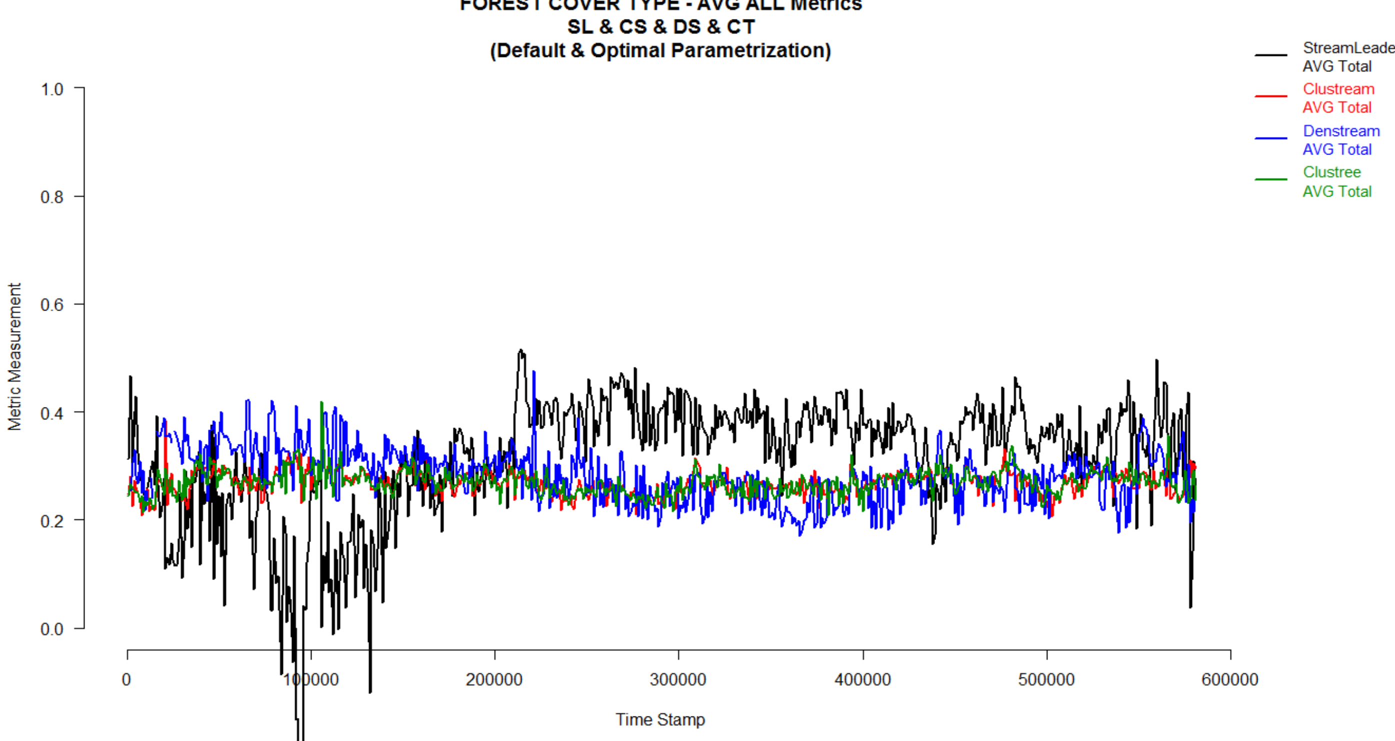


Figure 57: *Forest Cover Type*: overall quality performance

According to the quality metrics, it is a very difficult data set to cluster in streaming. *Homogeneity* renders very low quality for all algorithms except *StreamLeader*, which still suffers from a quality drop around in the middle of the experiment but then recovers. *Completeness* on the other hand shows weak performance for *StreamLeader*, values even drop out of the plot. In general, the classes in general seem to be fragmented in different pieces in several locations, so the algorithms struggle to find them using hyper-spherical clusters. Another factor of uncertainty is the way MOA calculates ground-truth and compare it with final clustering to generate external quality clustering measures. All in all, *StreamLeader* outperforms again in average *Clustream*, *Denstream* and *Clustree*. Table 18 displays average quality values obtained with default and optimal parametrization. While *Denstream* crashed in MOA delivering metrics in default parametrization, it outperformed *Clustream* and *Clustree* using optimal parameters, which could not improve their own results in any of the two parametrizations.

		Forest Cover Type			
		FEW BIG CLUSTERS			
		SLeader	CluSTR	DenSTR	CTree
DIM SPACE d=10	Default Param	**0.26**	**0.27**	**crash**	**0.27**
	Optimal Param	**0.37**	**0.27**	**0.36**	**0.27**

Table 18: *Forest Cover Type*: quality test results

3.4.2 Data Set 2: Network Intrusion

This data set was used for the Third International Knowledge Discovery and Data Mining Tools Competition, held together with KDD-99 The Fifth International Conference on Knowledge Discovery and Data Mining in 1999. It can be found frequently in stream clustering research, i.e. *StreamKM++* [AMR+12], *Clustream* [AHWY03], *Denstream* [CEQZ06], *D-Stream* [CT07] and *Clustree* [KABS11] among others. Table 19 contains a brief description of the dataset:

Real Data Set Name	**Network Intrusion (KDD-CUP'99)**
Size	4.8 million connections
Goal	The intrusion detector learning task is to build a network intrusion detector capable of distinguishing between *bad* connections, called intrusions or attacks, and *good* normal connections.
Description	Data contains computer connections, defined as a sequence of TCP packets starting and ending at some well defined times, between which data flows to and from a source IP address to a target IP address under some well defined protocol. Each connection is labeled as either normal, or as an attack, with exactly one specific attack type. Each connection record consists of about 100 bytes. It contains 42 total attributes (34 of those continuous used for clustering): duration, protocol_type , service , flag , src_bytes , dst_bytes , land , wrong_fragment , urgent , hot , num_failed_logins , logged_in , num_compromised , root_shell , su_attempted , num_root , num_file_creations , num_shells , num_access_files, is_host_login , is_guest_login , count , srv_count , serror_rate , srv_serror_rate , rerror_rate , srv_rerror_rate , same_srv_rate , diff_srv_rate , srv_diff_host_rate , dst_host_count , dst_host_srv_count , dst_host_same_srv_rate , dst_host_diff_srv_rate , dst_host_same_src_port_rate , dst_host_srv_diff_host_rate , dst_host_serror_rate , dst_host_srv_serror_rate , dst_host_rerror_rate , dst_host_srv_rerror_rate Cluster by type of connection in attack_types: normal or 24 attack types (i.e satan, smurf, spy, warezmaster, neptune, multihop , loadmodule , ipsweep , guess_passwd, buffer_overflow, etc)
Min classes[84]	1
Max classes[85]	5
Avg classes[86]	1.05
Location	`http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html`

Table 19: *Network Intrusion* data set details

We choose settings for the data stream (default values $decayHorizon = 1000$, $decayThreshold = 0.01$, $evaluationFrequency = 1000$) and normalize the data. We have to do it manually because MOA turns out to normalize but values are not restricted to range [0,1] with this specific data set. We then visualize the stream in MOA to have a feeling of the data. We quickly see that this type of data is special in the way that normal connections tend to be very similar in terms of duration, bytes transfered and so on. They are therefore placed in the same position or very close in the normalized d space, almost in min or maximum values of the normalized variable. The reason is that attacks, on the other hand, present sudden substantial variations in the variables (like rash increases in duration of connection, access files, root shell, etc), which take variables to the maximum or minimum values. Attacks also tend to be concentrated in a very short time span, which in a streaming scenario appears as a flash in the middle of *normal* connections. The algorithms are then confronted with very large periods of stability (*normal* connections) and sudden attacks lasting a fraction of that. An example of that would be, for instance, 40000 *normal* connections, combined with moments where 50 or 200 abnormal (*attacks*) take place. We confirm this with the information describing the data set in the table above, where *Min classes* = 1, *Max classes* = 5 (attacks), *Avg classes* = 1.05.

[84] Minimum number of different classes appearing simultaneously.
[85] Maximum number of different classes appearing simultaneously.
[86] Average number of different classes appearing simultaneously.

To better understand the explanations above and the data, Figure 58 shows what normal connections look like in MOA while streaming the data. Left and middle pictures show normal connections by visualizing two different pairs of attributes. Plot in the right, shows same set as plot in middle but displaying MOA's true clustering or ground-truth (contour in black). That is, MOA draws the smallest hyper-sphere it can in order to capture the class (all connections belonging to the same label). We notice that it is an over-sized true cluster and that its calculation in real data streams seems to pose a challenge to MOA sometimes:

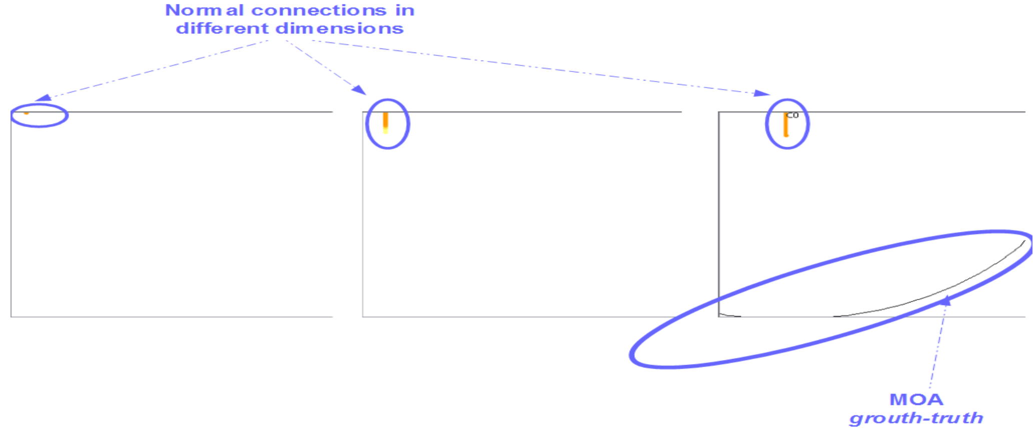


Figure 58: *Network Intrusion*: stream visualization using two attributes sets (left & middle) and MOA's true cluster (right)

We choose parametrizations for the algorithms ($horizon = 1000$ for all) as described in Table 20:

Algorithm	*StreamLeader*
Default Param	$D_MAX = 0.11$
Optimal Param	$D_MAX = 0.35$

Algorithm	*Clustream*
Default Param	- $MaxNumKernels = 100$ - $kernelRadiFactor = 2$
Optimal Param	- $MaxNumKernels$[87]$= 150$ - kernelRadiFactor=2

Algorithm	*Clustree*
Default Param	$maxHeight = 8$
Optimal Param	$maxHeight = 14$

Algorithm	*Denstream*[88]
Default Param	- $epsilon = 0.02$ - $beta = 0.2$ - $mu = 1$ - $initPoints = 1000$ - $offline = 2$ - $lambda = 0.25$ - $processingSpeed = 100$
Optimal Param	- $epsilon = 0.16$ - $beta = 0.2$ - $mu = 1$ - $initPoints = 1000$ - $offline = 2$ - $lambda = 0.25$ - $processingSpeed = 100$

Table 20: *Network Intrusion*: parametrization used for *StreamLeader*, *Clustream*, *Denstream* and *Clustree*

We can see an examples of *StreamLeader* (left side, contour in red) and *Clustream* (right side, contour in blue) delivering clustering at specific moments in Figure 59, together with true clustering from MOA (contour in black). Upper plots show *instant 1* in the stream, where *StreamLeader* seems to adjust well to the set of points while *Clustream* delivers only a small clustering not covering all the connections. Bottom plots show another time *instant 2*, where *StreamLeader* creates a *LeaderKernel* that still seems to capture well the connections, while *Clustream* delivers again a very small cluster.

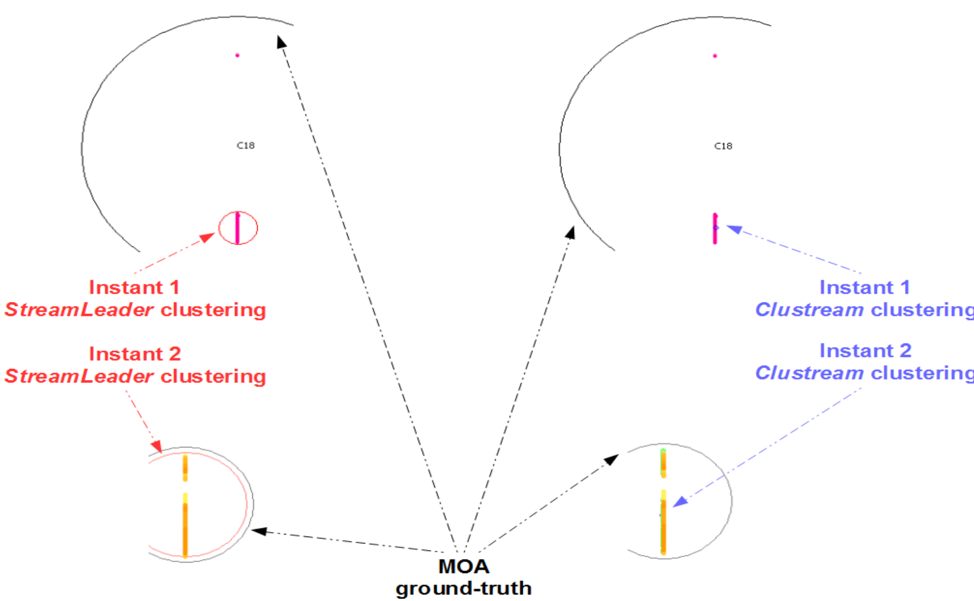


Figure 59: *Network Intrusion*: clustering at two instants by *StreamLeader* (red) and *Clustream* (blue)

Figure 60 displays a very common situation that will explain why this time *Denstream* will deliver much better quality results than what we saw in other sections of this work. Very similar connections imply very similar placing of the instances in d space, lasting also a considerable amount of time since normally there are no cyber-attacks. Because there are no shapes to form, *Denstream* concentrates its clustering in very small areas, which almost map the instances themselves, achieving therefore higher scores. *StreamLeader* (red contour) creates a *LeaderKernel* capturing the connection, which is bigger than what seems necessary in this set of attributes visualized. The *LeaderKernel* does not contract further even if it seems natural to do so, probably because other dimensions show instances not concentrating so much[89]. *Clustream* and *Denstream* deliver a much smaller cluster.

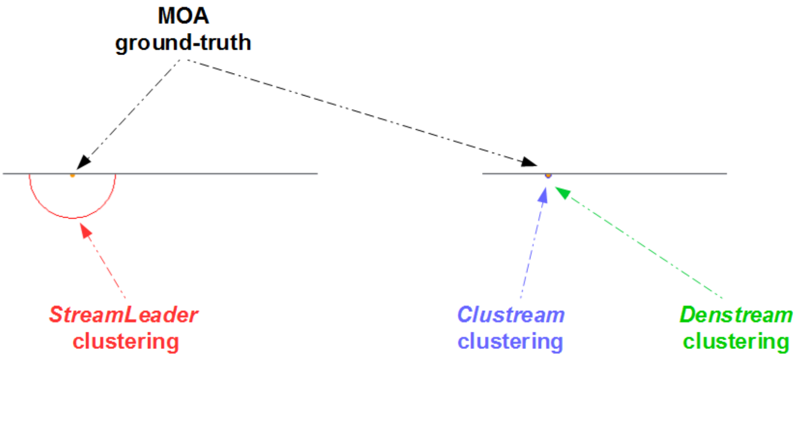


Figure 60: *Network Intrusion*: clustering by *StreamLeader* (red), *Clustream* (blue) and *Denstream* (green)

[87]While 100 micro-clusters achieves *micro-ratio* ≥ 10, we still opt to increase that ratio to achieve finer granularity.
[88]We recall that we could not replicate the author's preferred parametrization for *Denstream* in MOA.
[89]We found time consuming and difficult finding the right set of attributes to display. Advances techniques for visualizing clustering in streaming scenarios is an evident need for the future.

Finally, Figure 61 shows a sudden and short-lived *satan* attack (in purple), which appears suddenly and disappears very quickly. It contains two screen-shots with two time instants (*instant 1* top plots where attack just appears and *instant 2* bottom plots, where the attack crossed the screen and banishes). We can see how *StreamLeader* (plots in the left, red contour) and *Clustream* (plots in the right, blue contour) react to the attack (fast change in data distribution). *StreamLeader*, at *instant 1*, covers initially the data with one *LeaderKernel* and suddenly tries to adjust its size to the very fast moving sequence of connections, producing the effect we observe of co-center spheres. At *instant 2*, a big unique *LeaderKernel* captures the whole tract of the attack. On the right side, *Clustream* also adjusts its micro-clusters as fast as it can at *instant 1*. At *instant 2* it produces two small clusters. We can also observe true clustering (black contour) in MOA, producing the bigger cluster of all. Due to the fast pace changes produced in this sort of attacks, we expect abrupt changes in the quality metrics.

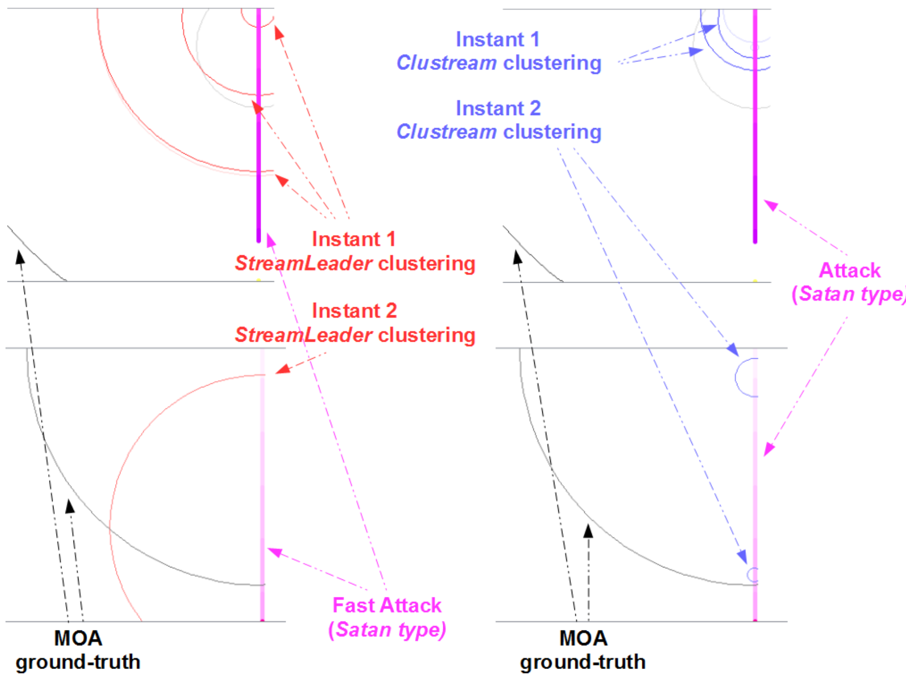


Figure 61: *Network Intrusion*: Connection attack and reaction of *StreamLeader* (red) and *Clustream* (blue)

We gather all quality metrics for both default and optimal parametrization in Figures 62 *CMM*, 63 *Silh Coef*, 64 *F1-P*, 65 *F1-R*, 66 *Homogeneity*, 67 *Completeness*, 68 *Rand Statistic*, 69 *Q_AVG*:

Figure 62: *Network Intrusion*: *CMM* on default vs optimal parametrization

Figure 63: *Network Intrusion*: *Silhouette Coefficient* on default vs optimal parametrization

NETWORK INTRUSION - F1 Precision
SL & CS & DS & CT
(Default Parametrization)

Metric Measurement
1.0
0.8
0.6
0.4
0.2
0.0
0
1000000
2000000
3000000
4000000
5000000
Time Stamp

StreamLeader AVG F1P DEFAULT
Clustream AVG F1P DEFAULT
Denstream AVG F1P DEFAULT
Clustree AVG F1P DEFAULT

NETWORK INTRUSION - F1 Precision
SL & CS & DS & CT
(Optimal Parametrization)

Metric Measurement
1.0
0.8
0.6
0.4
0.2
0.0
0
1000000
2000000
3000000
4000000
5000000
Time Stamp

StreamLeader AVG F1P OPT
Clustream AVG F1P OPT
Denstream AVG F1P OPT
Clustree AVG F1P OPT

Figure 64: *Network Intrusion*: *F1-P* on default vs optimal parametrization

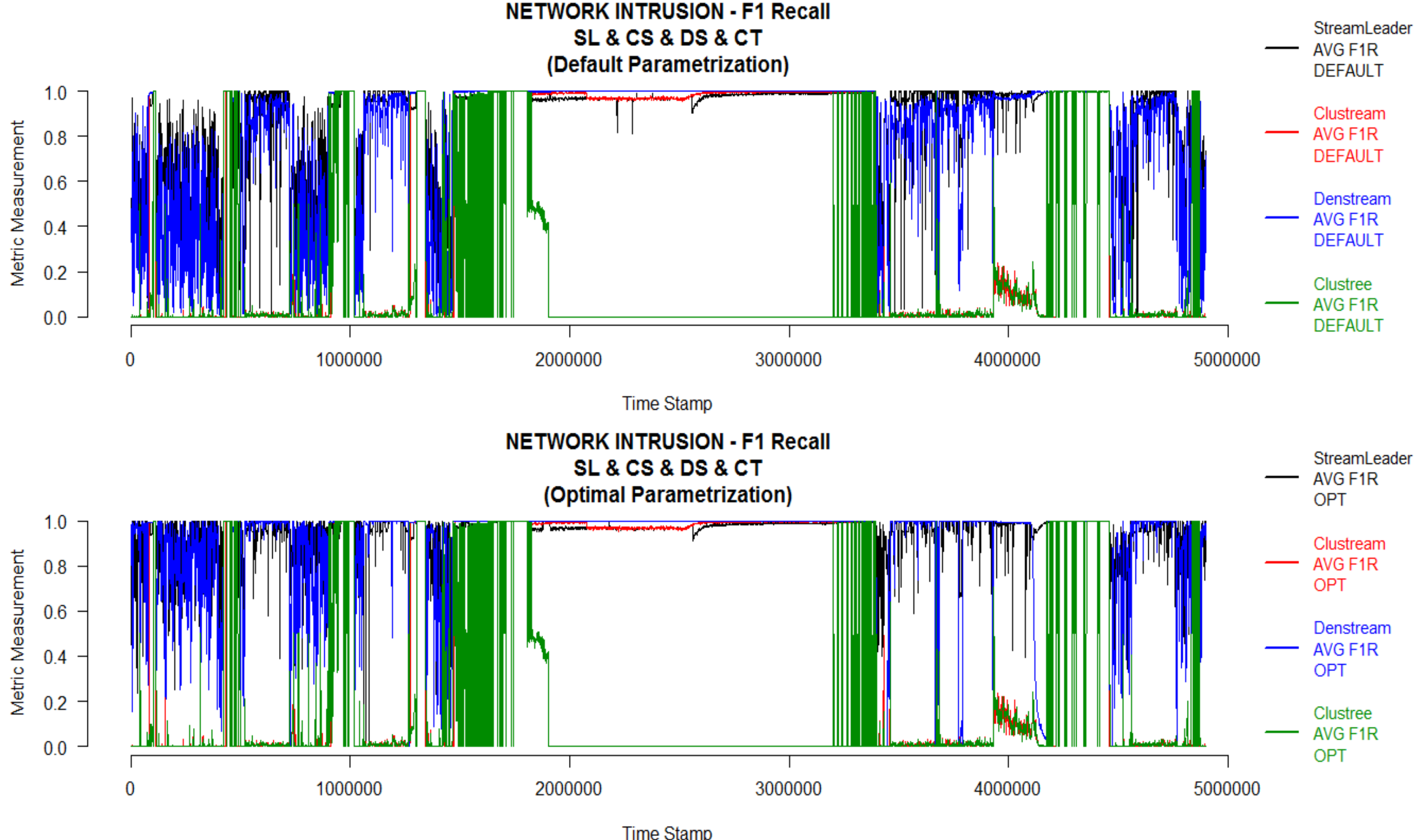


Figure 65: *Network Intrusion*: *F1-R* on default vs optimal parametrization

Figure 66: *Network Intrusion*: *Homogeneity* on default vs optimal parametrization

Figure 67: *Network Intrusion*: *Completeness* on default vs optimal parametrization

Figure 68: *Network Intrusion*: *Rand Statistic* on default vs optimal parametrization

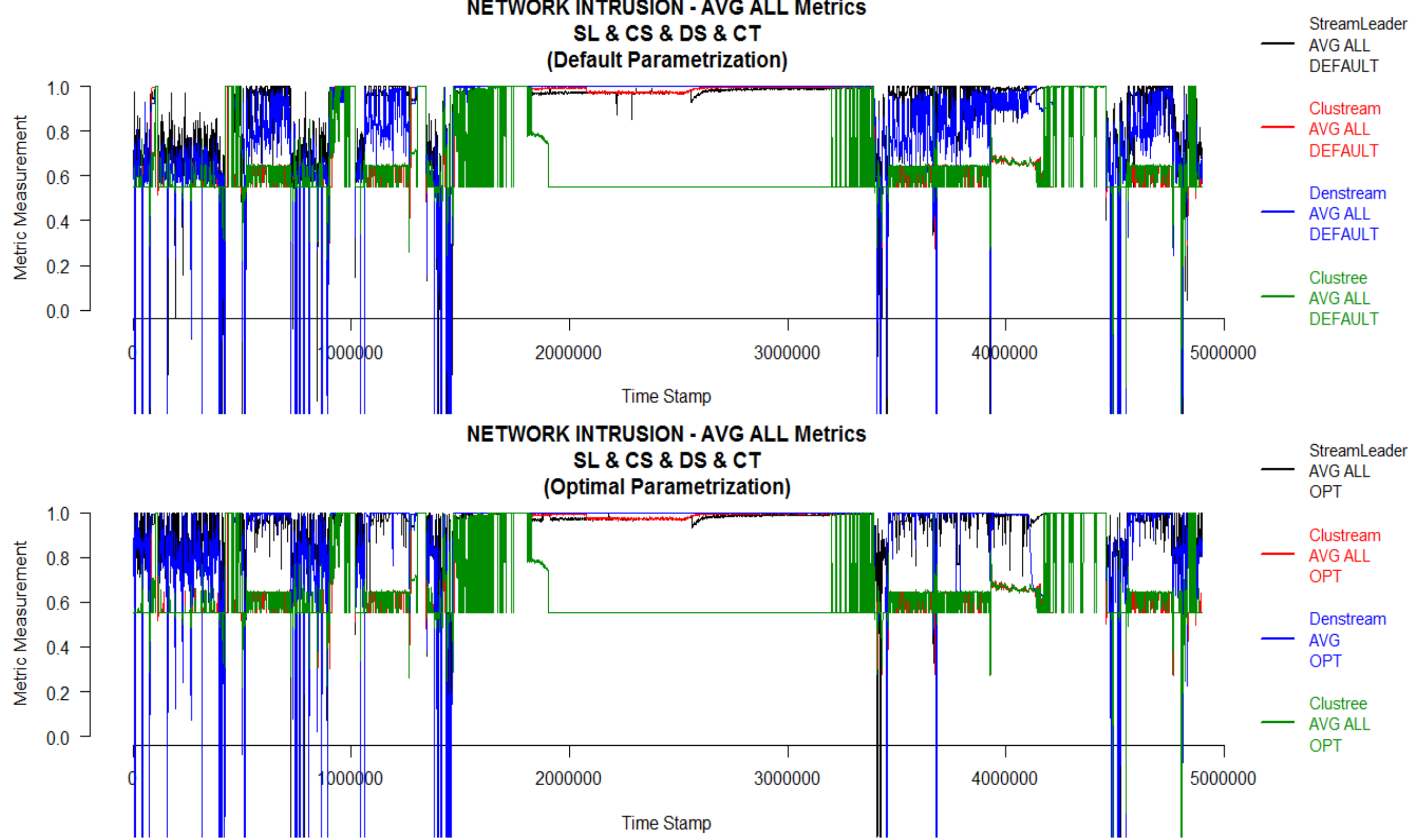


Figure 69: *Network Intrusion*: overall performance on default vs optimal parametrization

Together with the overall performance Q_AVG with both parametrizations in Figure 70:

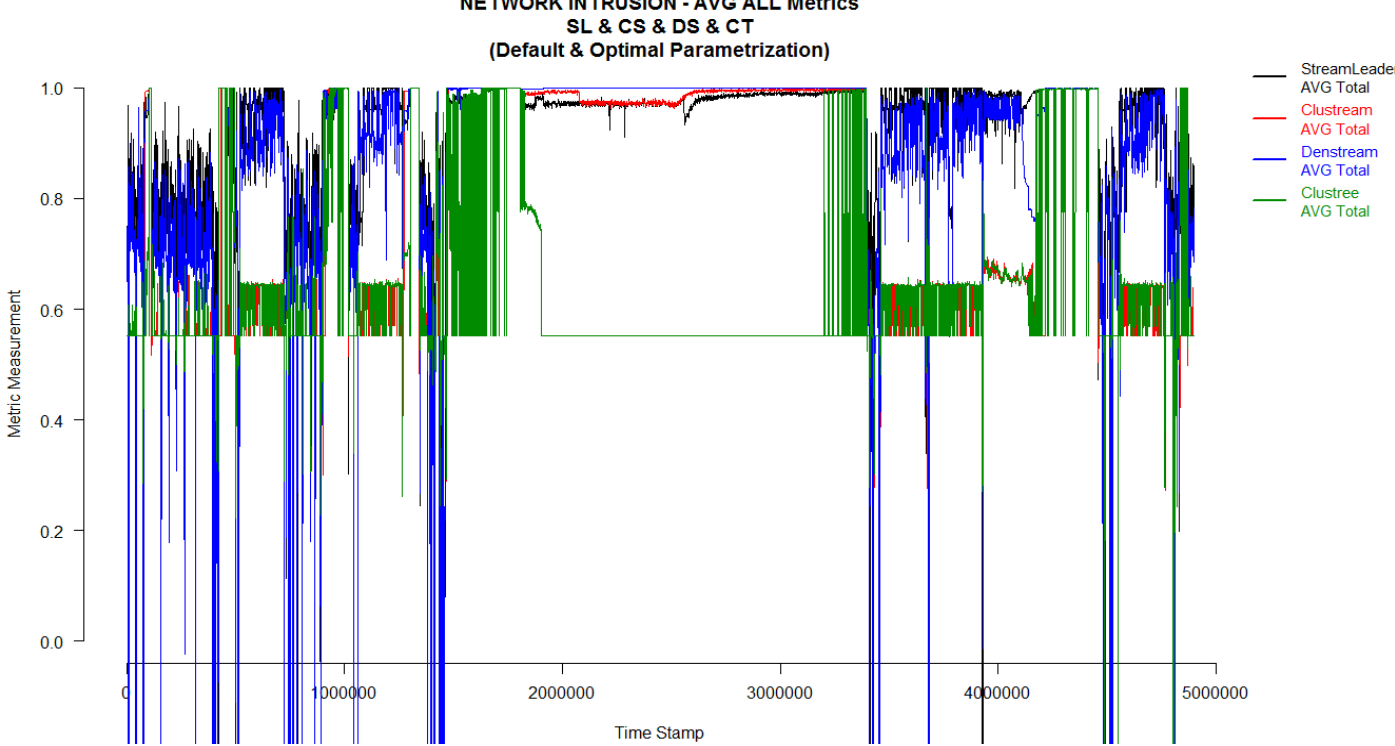


Figure 70: *Network Intrusion*: overall performance of *StreamLeader*, *Clustream*, *Clustree* and *Denstream*

As we see in the plots, the changes in metrics are very abrupt due to the nature of the attacks. The fact that they are very short in duration produces the sudden drops in some of the metrics. *Completeness* in Figure 67 seems to be a very special case in experiencing the biggest decreases in quality. The reason is that attacks occupy a place in the d space, away from *normal* connections, and *StreamLeader* can normally detect them. The most difficult situation for the clustering task arises when the attacks come in the form of fast-moving instances in the d space. Then, *LeaderKernels* must track the *moving target* and adjust constantly. In this cases, *Completeness* drops substantially because all members from the same class can almost never be allocated to the same *LeaderKernel* which takes a certain amount of instances to reallocate (move) its *leader*. Looking at the values that MOA delivers for *Completeness*, we also observe extremely large negative values (i.e -8), even if the measure should be bounded within range [0,1][90]. Another factor could also be the way true clustering or ground-truth is calculated *on-the-fly* in MOA, because matching against it is necessary for external measure calculation.

This problem immediately draws our attention to the fact that *horizon* is a key piece in this sort of environment with streams having large periods of stability (*normal* connections) and sudden and abrupt changes (*fast attacks*). In normal situations, as we mentioned in former sections, a trade-off between stability/efficiency of the algorithms and rapid response is necessary. MOA allows *horizon* specification, but it is fixed, in this case for the 4.8 million stream instances. A technique like *ADWIN* or *ADWIN2*, as explained in [BG06] would be perfect for this data set to reduce *horizon* automatically, recalculate *LeaderKernels* faster and and give a more accurate response to attacks.

Since 5.8 million instances overcrowd the plots, we also show separately the Q_AVG overall metric for each algorithm in Figure 71 and Figure 72. We observe how *StreamLeader* and *Denstream* suffer from intense drops but achieve an overall higher quality than *Clustream* and *Clustree*.

[90] Analyzing the functioning of the measure in MOA is out of scope of this work.

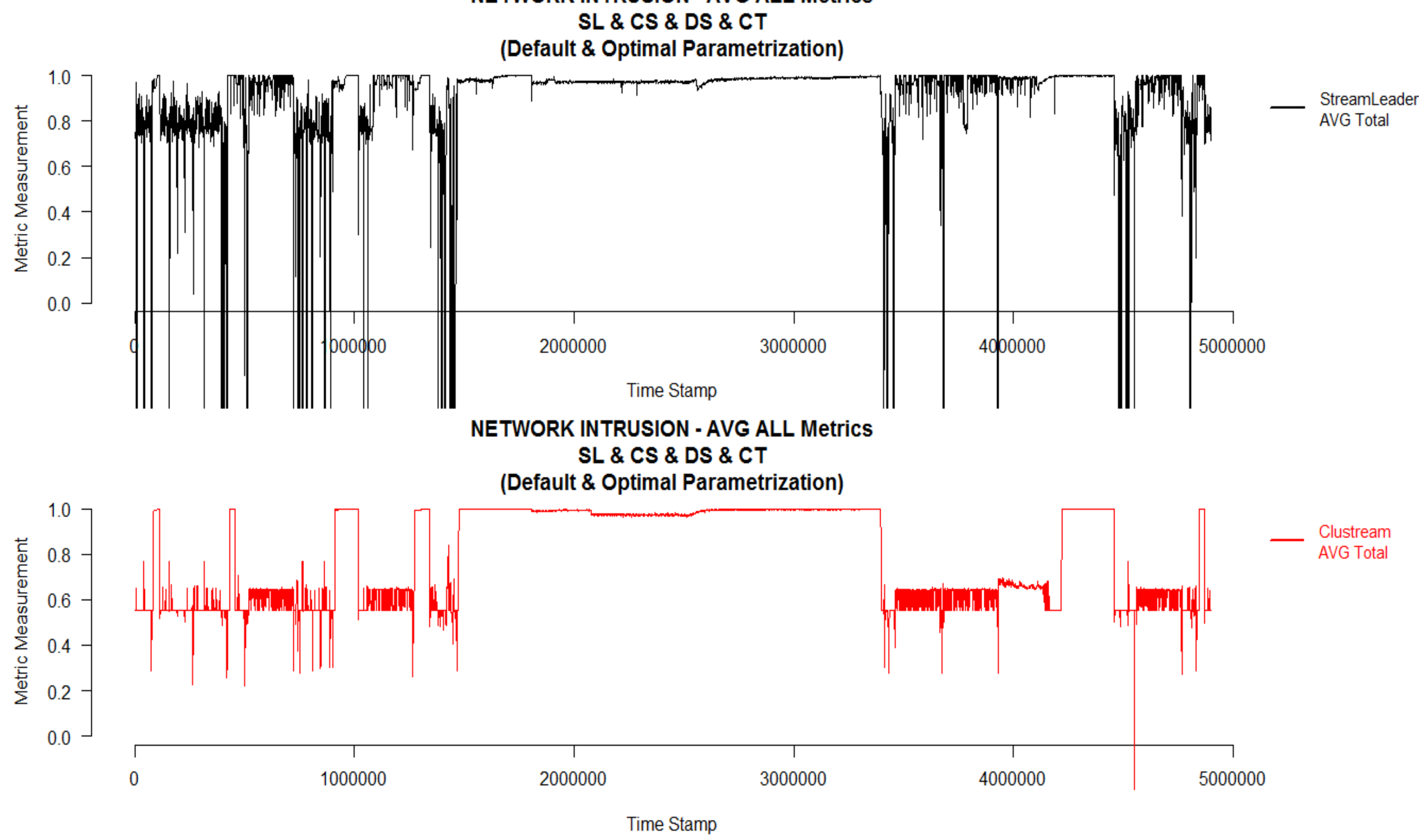


Figure 71: *Network Intrusion*: overall performance of *StreamLeader* and *Clustream*

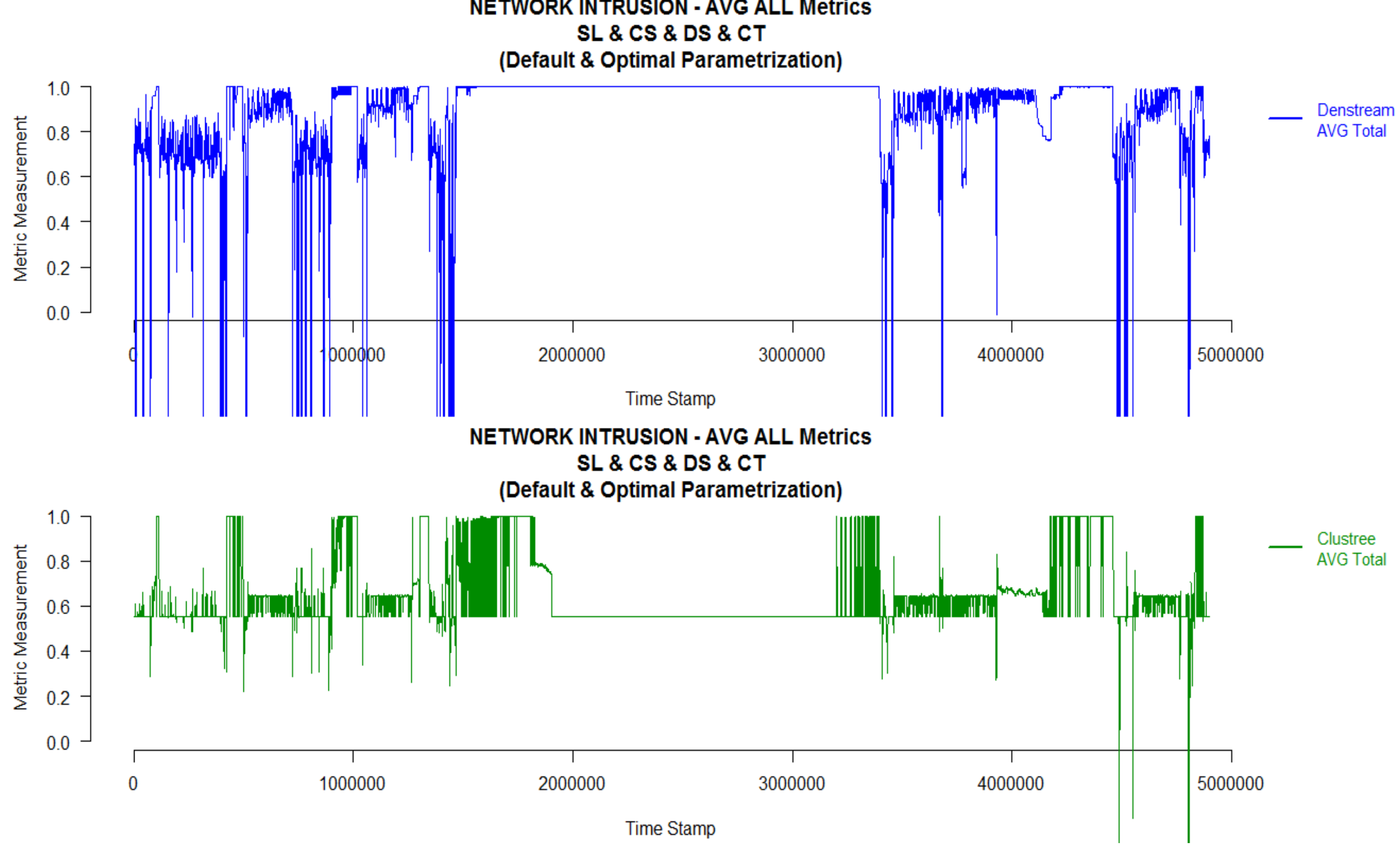


Figure 72: *Network Intrusion*: overall performance of *Denstream* and *Clustree*

To investigate further the drops in quality, we look with detail at metrics delivered *StreamLeader* only, shown in Figures 73, 74, 75 and 76. We want to trace where the sudden drops in quality originate. We see in Figure 75 that they are due to *Completeness*, which experiences decreases in quality reaching values of up to -8, even when the metric should be constrained to [0,1] in MOA. Such high negative values drag heavily the average quality for those instants.

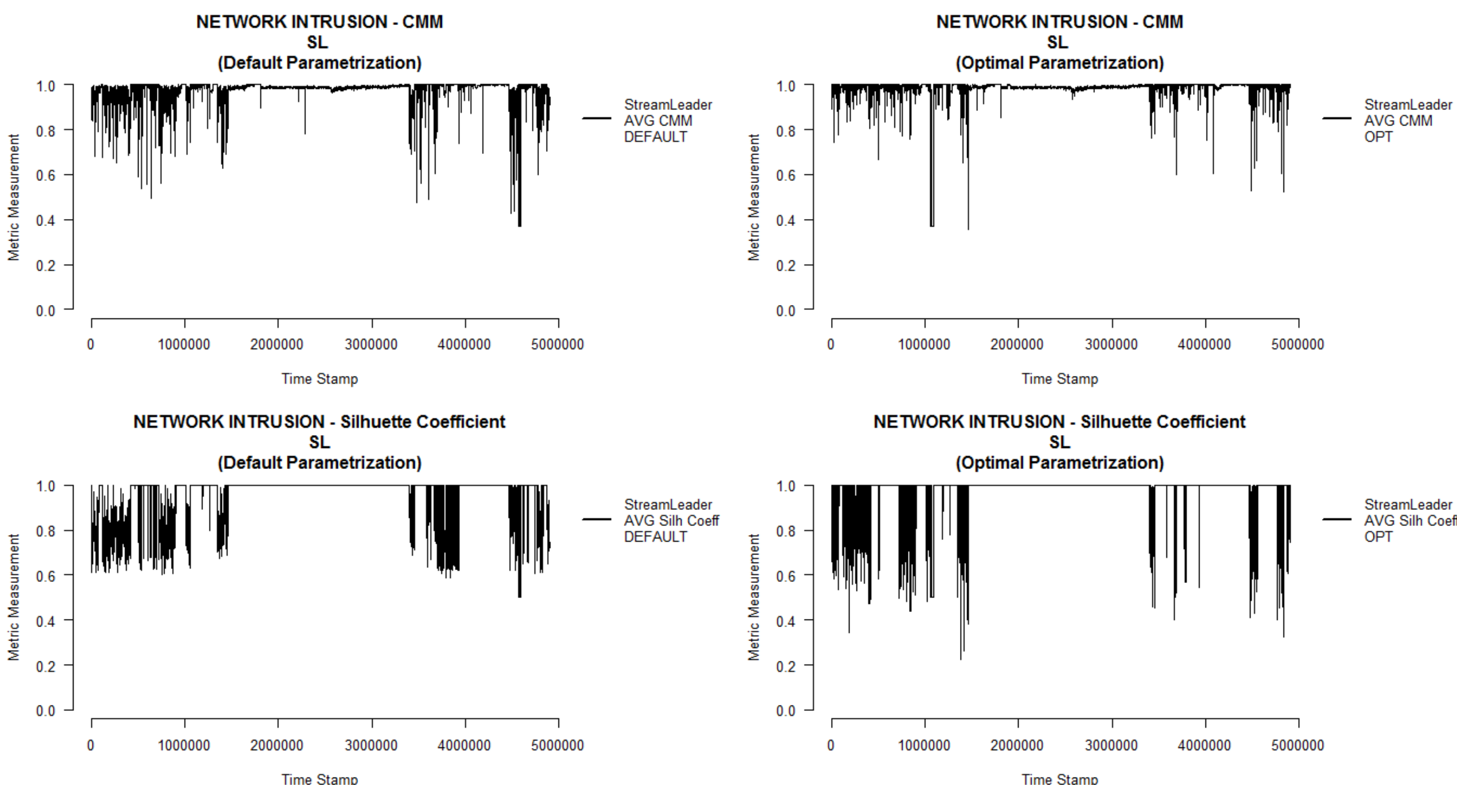


Figure 73: *Network Intrusion*: *CMM* and *Silhouette Coef* for *StreamLeader*, default vs optimal parametrization

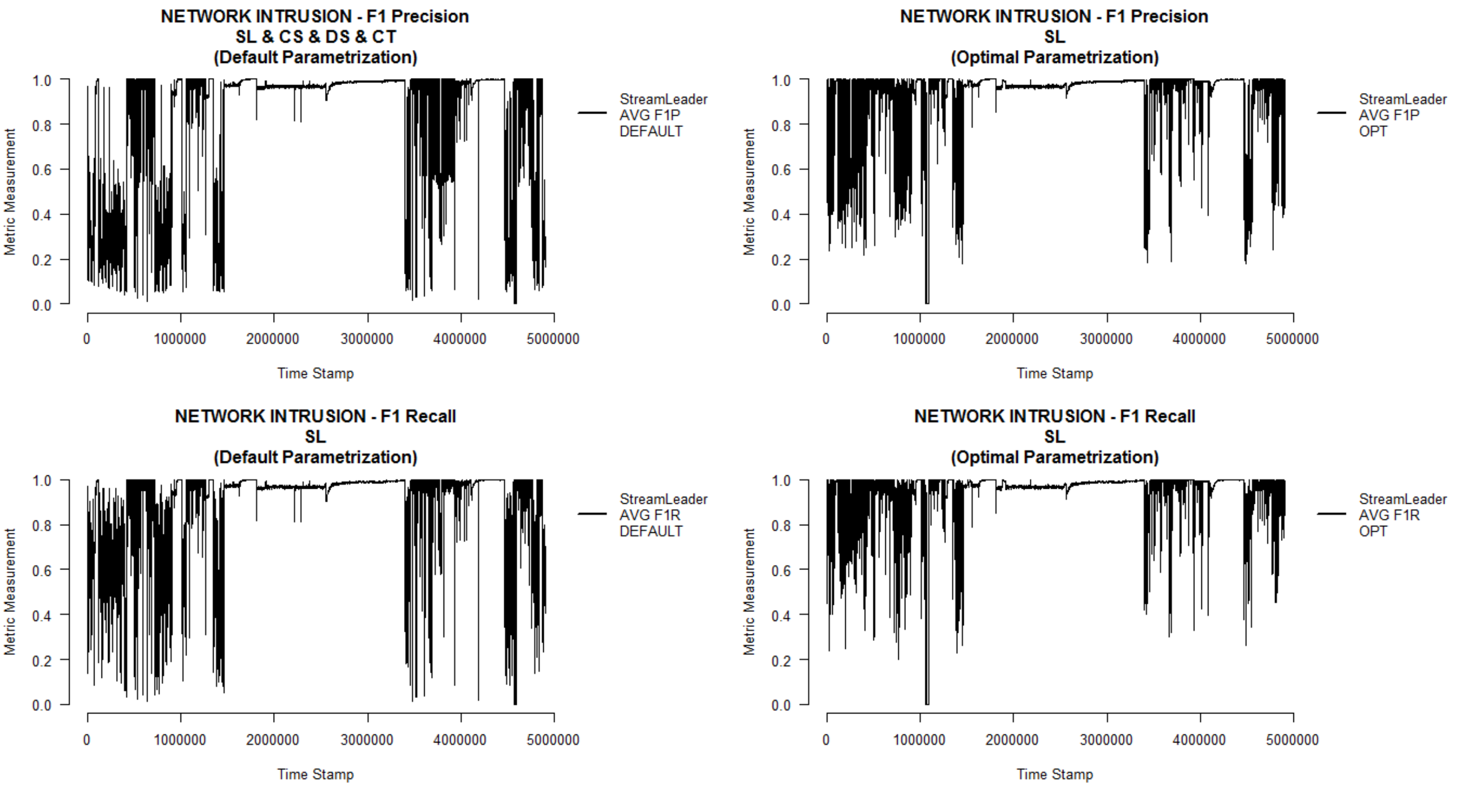
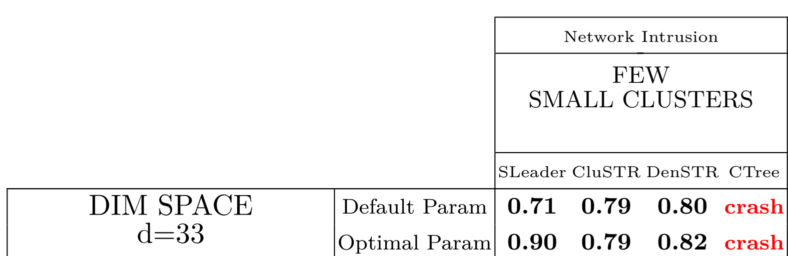


Figure 74: *Network Intrusion*: *F1-P* and *F1-R* for *StreamLeader*, default vs optimal parametrization

Figure 75: *Network Intrusion*: *Homogeneity* and *Completeness*, default vs optimal parametrization

Figure 76: *Network Intrusion*: *Rand Statistic* and Q_AVG for *StreamLeader*, default vs optimal parametrization

Table 21 contains the numeric results of the tests. MOA crashes while producing *Clustree*'s measures. Ten values are set as NA (*Not Available*). If we were to assign default of 0.5 to those, *Clustree* would deviver average value of 0.65 and 0.65 for default and optimal parametrization respectively. *StreamLeader* outperforms again, in average, *Clustream*, *Denstream* and *Clustree*



		Network Intrusion — FEW SMALL CLUSTERS			
		SLeader	CluSTR	DenSTR	CTree
DIM SPACE d=33	Default Param	**0.71**	**0.79**	**0.80**	**crash**
	Optimal Param	**0.90**	**0.79**	**0.82**	**crash**

Table 21: *Network Intrusion*: quality test results

As side effect, we could use *StreamLeader* even as an attack warning system. In best case scenario, the connections of an attack form a cluster (or clusters) that can be tracked in *d* space properly. If they reallocate too fast for our time window to track them efficiently, then we could theoretically detect those changes and adjust to a smaller *horizon*. Even if we could not adjust it, attacks trigger in *StreamLeader* the creation of *LeaderKernels* to try capturing them. With proper D_MAX, suddenly we detect attacks. Figure 77, upper plot displays number of classes in the data set as they appear in the stream (that is, classes = connection types, so that several connection types implies a *normal* and several *abnormal* connections). Bottom plot contains number of *LeaderKernels* maintained by *StreamLeader* at those specific moments. Above a certain threshold, we get strong indications that connection attacks are taking place.

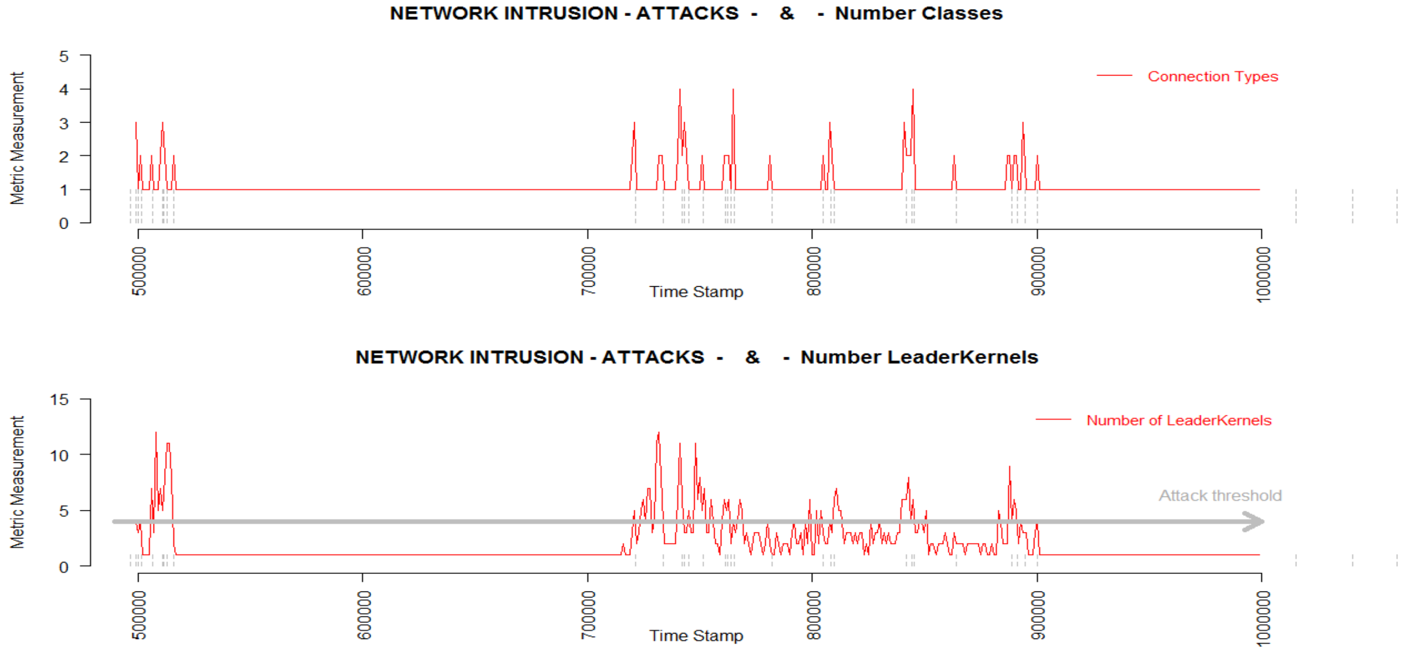


Figure 77: *Network Intrusion*: using *StreamLeader* as an attack warning system

3.5 Scalability and sensitivity tests

Regarding the aspect of theoretical complexity in stream clustering algorithms, we review the literature and find that a comprehensive theoretical analysis on space and/or performance complexity seems not to be standard in published papers. *Clustream* [AHWY03], *Denstream* [CEQZ06], *Clustree* [KABS11], *D-Stream* [CT07], *StreamKM++* [AMR+12], among others, favor runtime performance and scalability testing instead. We therefore follow this approach and design scalability and sensitivity testing focusing on:

- number of clusters vs dimensionality, using default parametrization
- number of clusters vs dimensionality, using optimal parametrization
- number of clusters vs number of instances, using default parametrization

We use MOA command features to run the tests in console. Basic settings are: $instanceLimit = 500000$ (stream size 500000 instances), $measurecollectiontype = 0$ (no measures collected), $numClusterChange = 0$ (no change in number of clusters), $KerneRadius = 0.025$ (we keep clusters small so that we avoid crashes in execution in high dimensional space[91]), $RadiusChange = 0$ (cluster radius does not change), $Noise = 0.1$ (10% noise), $EventFreq = 50000$ (one event every 50000 instances), $MergeSplit = N$ and $DeleteCreate = N$ (disabling events). We generate tests by changing $numClus =?$ (number of clusters) and $numAttr =?$ (dimensionality). An example of such command to test *StreamLeader*, $D_MAX = 0.04$, RBF random generators to create the stream, 50 clusters of radius 0.025, disallow events creation/deletion, merging/splitting of clusters, (in normalized d space), 50 dimensions and half a million instances:

java -javaagent:sizeofag.jar moa.DoTask "EvaluateClustering -l (StreamLeader.StreamLeader -d 0.04) -s (RandomRBFGeneratorEvents -K 50 -k 0 -R 0.025 -n -E 50000 -a 50) -i 500000"

We test each of the four algorithms against 2, 5, 20, 40 and 50 clusters, in spaces of dimensionality 2, 5, 20 and 50. This results in 20 tests per algorithm, so 80 tests for all four algorithms. Each test in executed 10 times to get average results, totaling 800 tests producing the average results in Figure 78:

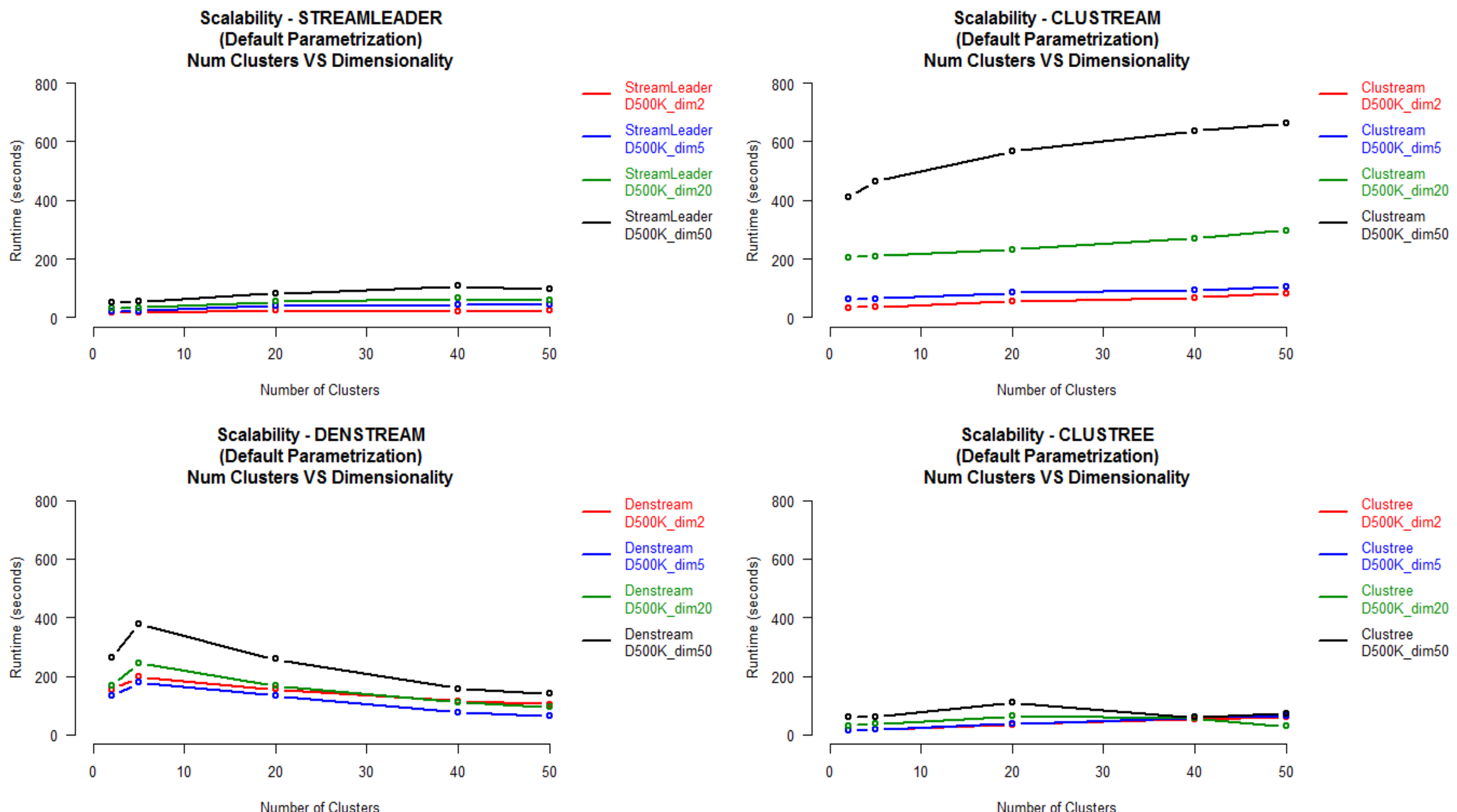


Figure 78: Scalability: number of clusters VS dimensionality, (default parametrization)

[91] In dimension $d = 50$, we had to reduce the cluster radius down to 0.02 because MOA needs small clusters if they are to be created in such high dimensional space.

StreamLeader scales well in terms of number of clusters, linearly and with very small slope. Also, in terms of dimensionality, where it handles 50 dimensions with very small burden in runtime performance. This can be explained by the fact that *StreamLeader* is designed to keep numbers of *LeaderKernels* to the minimum, by attacking aggressively noise. Then, computations only grow marginally with high dimensions (the vectors of size d that keep summarizations are also kept to a minimum). Having few *LeaderKernels* also implies keeping memory free and reducing distance computations among the small number of *leaders*. Lastly, we had also eliminated the use of a conventional clustering algorithm, which normally implies the firing of several rounds (100 in the case of *Clustream*) of k-means with different seeding. *Clustream* has evident problems when it moves into 20 and 50 d. Its micro-clusters need more maintenance in such spaces plus the firing of the clustering. *Denstream*'s performance follows a special behaviour, probably due to the lack of proper parametrization, so comments on its performance would be probably not well founded. *Clustree* seems to scale remarkably well in d up to 20, in line with *StreamLeader*. This is thanks to its design using tree structures, where closest *CF*s are found in logarithmic time. Therefore processing of instances happens very fast, with reduced number of distance calculations. Also, clustering is delivered by returning the *CFs* placed in the leafs of the tree, which also does not hinder performance. However, we notice that its runtime performance drops in higher d (starting from 20 on), which does not make sense. We do not have visibility on the results, but a strong indication comes from the fact that *Clustree* crashed 5 times while generating the quality metrics shown in quality testing, all in d 20 and 50, default parametrization and many or medium amount of clusters (*scenario* 3, 4, 5, 8, 9, 10). So this drop could well be because *Clustree* crashes but MOA somehow still measures time until the end of the half a million instances.

Since real data is multi-dimensional in nature and we see that dimensionality poses a real challenge to the algorithms, we decide to compare each in a 20d space with 500000 instances, this time focusing on default vs optimal parametrization,[92] and perform each test 10 times, obtaining the results shown in Figure 79:

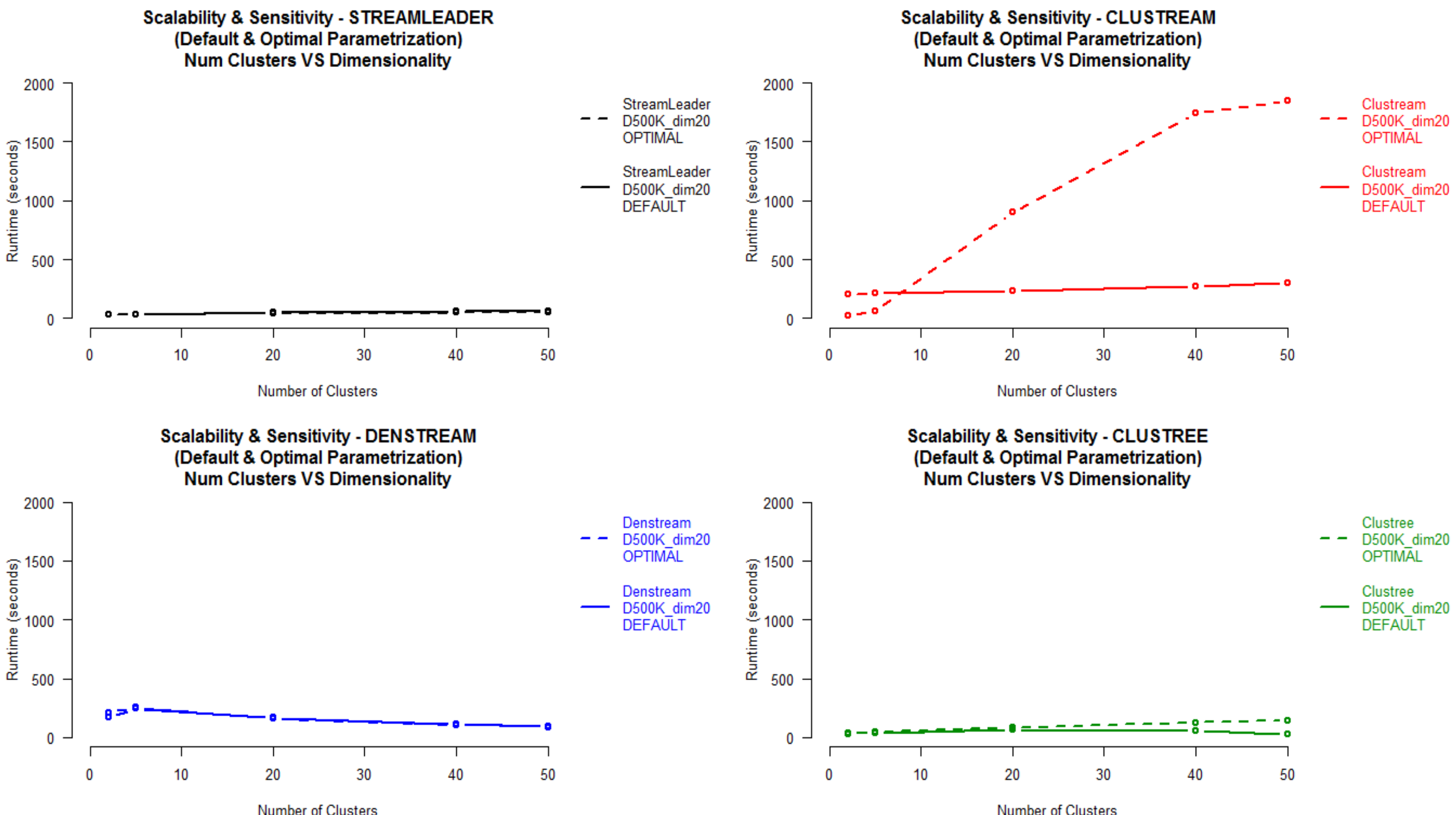


Figure 79: Scalability & sensibility: number of clusters VS parametrization (default vs optimal) in 20d

Clustream shows how optimal parametrization degrades its performance substantially. We reduce the scale in Figure 80 to differentiate the behavior of the others:

[92]In line with author's guidelines as explained in former sections.

Figure 80: Scalability & sensibility: number of clusters VS parametrization (default vs optimal) in 20d, reduced scale

And plot them together per type of parametrization in Figure 81:

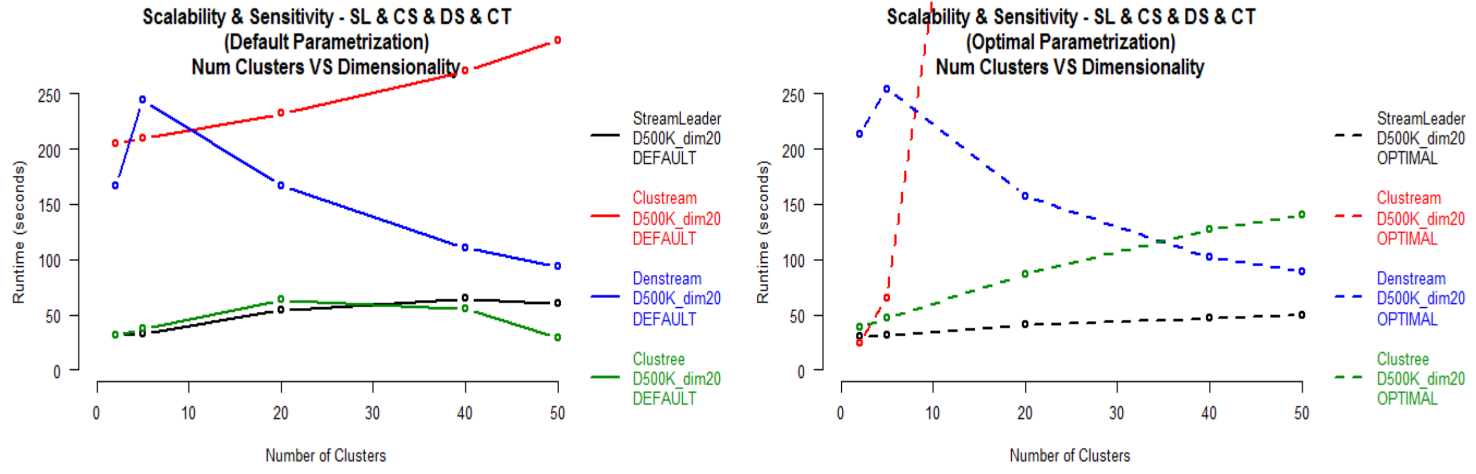


Figure 81: Scalability & sensibility: number of clusters VS parametrization (default & optimal) in 20d, reduced scale (2)

We can draw some conclusions with the small size cluster setting. *StreamLeader* manages to even improve runtime performance when optimal parametrization is chosen. This can be explained by the fact that adjusting the radius close to the size of the true biggest cluster causes fewer collisions, also occurring not so often in higher d spaces so the *StreamLeader* spends most of its time contracting to the correct radius. With default parametrization, probably D_MAX is too large, therefore collisions with other *LeaderKernels* might occur, triggering then the merging operations, which implies computation over the always restricted set of *LeaderKernels*. Still, it handles numerous clusters well using both parametrizations. *Clustream* pays a price for optimal tunning, having to maintain its *micro-ratios* with numerous micro-clusters in high d. *Clustree*

increases runtime performance with a higher slope than *StreamLeader*, avoiding this time the crashes it experiences with default parametrization. Still, it outperforms *Clustream* clearly. This is again probably due to the logarithmic access to the right *CF* when a new instance comes plus its internal handling of the tree and the fact of delivering clustering just by returning the *CF*s located in the leafs.

Lastly, we test again scalability by increasing number of instances in the data stream being launched at the algorithms. We create data streams of different lengths (K=1000 instances) 100K, 250K, 500K, 1000K, 2000K), in 20*d* space, using 5, 20 clusters and default parametrization. Rest of the configuration remains the same as in former synthetic tests, resulting in Figure 82:

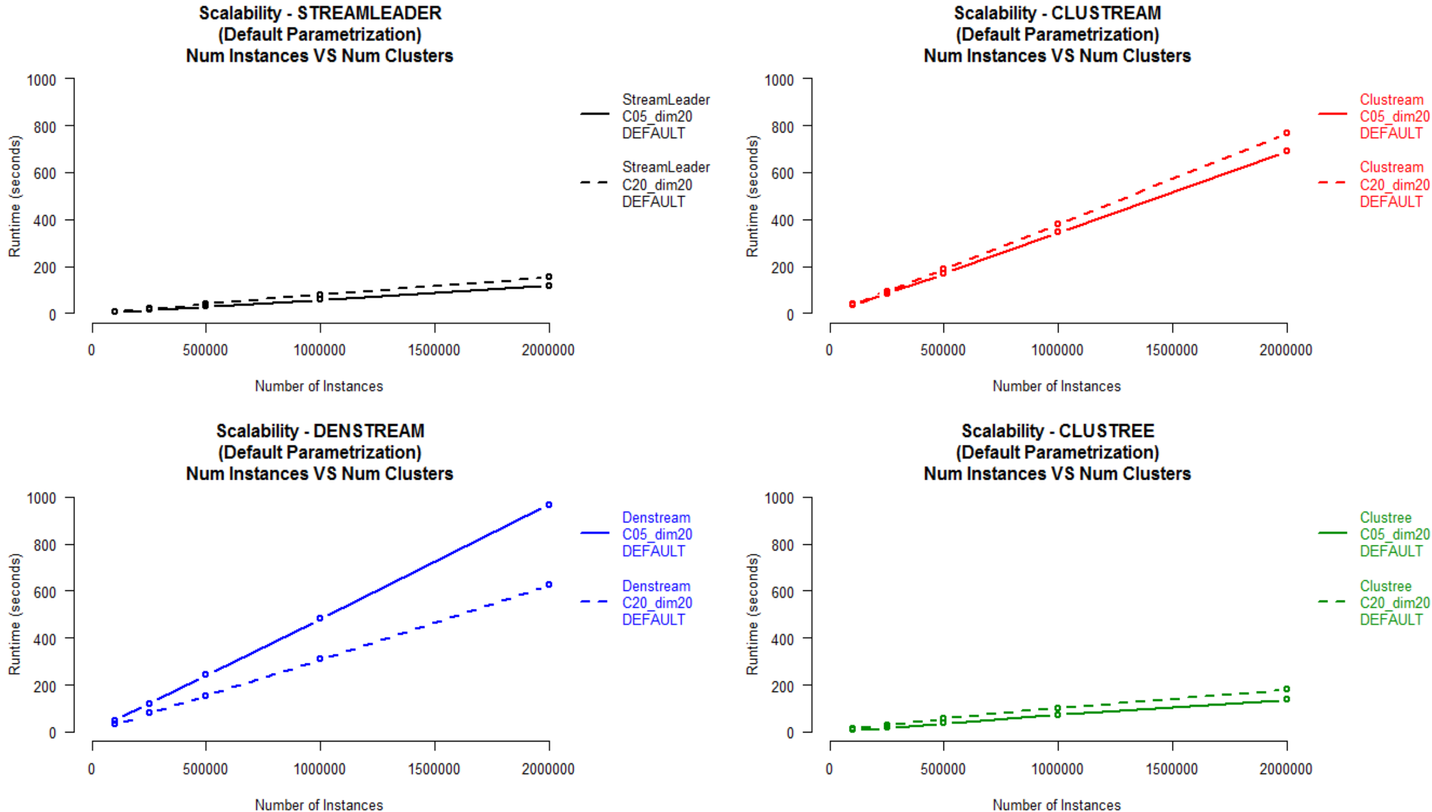


Figure 82: Scalability: number of clusters VS number of instances, in 20*d*, default parametrization

StreamLeader scales again linearly with a small slope, increasing its runtime performance slightly when the stream contains 20 clusters. This means that no bottleneck occurs, and that the *LeaderKernels* absorb the 2 million instances effectively (in less than 3 minutes). *Clustree* also handles the increasing stream efficiently, while *Clustream* seems to scale worse, taking around 12 minutes to process the data stream.

4 Part 4 - Conclusions and future work

4.1 Conclusions

In this thesis, we have developed a fast and accurate algorithm for clustering evolving streams of data, named *StreamLeader*, as it is based on the principles of the *Leader* clustering algorithm defined by John Hartigan in [Har75].
Our work needed to start with a review of the state of the art techniques in stream clustering. We came to the conclusion that certain drawbacks are implicit to the design of most of the existing solutions. They use a combination of online abstractions of the stream and offline phases, where resulting micro-clusters from online step are used offline as pseudo-points in a conventional clustering algorithm. First, the use of conventional clustering in a streaming problem, does not seem entirely desirable. Second, micro-clusters tend to grow bigger in size in higher dimensional spaces, since data tends to be sparse, which implies that final clustering also tends to be over-sized and overlapping. And third, noise can pose serious challenges to such approaches because it *contaminates* the micro-clusters and therefore the final clustering using them as basis.
These considerations gave us a clear idea of what we wanted to avoid and what to achieve with the *StreamLeader* in a streaming scenario, while bearing in mind the characteristics we had to keep to follow *leader* principles. Extending the concept of cluster feature vectors gave us an effective way to abstract parts of the stream and implement a functioning sliding window model, allowing *LeaderKernels* to age. We gave them capability to contract and expand and made them very efficient in capturing groups of instances in the multi-dimensional space based on a normalized proximity measure (distance) we developed. The use of these new units, called *LeaderKernels* proved very effective in encapsulating and capturing *concept drift* in the stream. With them, we could *absorb* statistically the stream, but we needed to counter the tendency of *leader-based* algorithms of filling the space with *LeaderKernels*. The solution to this problem gave us the key for a more ambitions and integral solution, which avoids the need of a conventional clustering algorithm in the offline phase and that controls aggressively noise before it can contaminate resulting clustering. We designed a novel approach to attack noise in the distribution of *LeaderKernels*, comprising of a combined application of percentile and logarithmic cuts, aiming at noise located in the tail and in the elbow of the distribution respectively.
With regards to testing, we realized that it is common in the literature to use certain quality metrics which are not optimal, due to either a tendency to deliver overoptimistic results (i.e purity) or because they lack normalized results (i.e SSQ). Since individual metrics often focus on a partial view of the clustering quality, we deemed necessary to build a sound metric and did so by combining seven well-known clustering quality measures, offering in this way a more comprehensive one. A framework to allow proper comparison of streaming solutions was also needed, so we put together 80 different use case configurations for synthetic testing, including noise levels, dimensionality and stream composition. Real data testing was also needed to put *StreamLeader*, *Clustream*, *DenStrean* and *Clustree* to the test. The results were encouraging, achieving and surpassing the initial expectations for the *StreamLeader*. Below the key milestones achieved:

1) We created *StreamLeader*, a new stream clustering algorithm based on the *leader* principles, and specialized in detecting hyper-spherical clusters in multi-dimensional space.
2) We integrated *StreamLeader* in MOA streaming platform.
3) *StreamLeader* uses an alternative novel approach in order not to use any conventional clustering algorithm in the offline phase (which most of the other solutions do). It is based on percentile and logarithmic cuts on the distribution of *LeaderKernels*, combined with contraction/expansion capabilities for cluster detection.
4) To measure clustering quality, we avoided the use of single metrics (which is the most frequent case in the literature) and created a sound metric which offers a more solid view of quality results.
5) To test quality results, we designed a comprehensive set of 80 synthetic scenarios and also used real data to test the algorithms with multiple configurations and uniformly.
6) *StreamLeader* outperformed, in terms of quality results, *Clustream*, *Denstream* and *Clustree*, in noise tolerance, dimensionality handling and stream composition.
7) *StreamLeader* scaled remarkably well, in terms of cluster number, stream size, dimensionality and parametrization, outperforming in runtime performance *Clustream*, *Denstrean* and *Clustree* in most cases.
8) *StreamLeader* achieves its performance with only one *user-friendly* parameter (*Clustree* 1, *Clustream* requires 2 and *Denstream* 7, in MOA).

4.2 Future work

As future work, there are plenty of areas where we could work to make *StreamLeader* more effective:

1) Even if completely 0% noise environment is not very likely, a new version will handle automatic adjustments for the percentile cuts, which could be entirely disabled when necessary.

2) One important improvement would be to make the algorithm react faster to sudden changes in underlying data distributions. The need for this was clearly visible while testing *Network Intrusion* real data set when short-lived attacks occurred suddenly, making it hard for the algorithms to adjust fast enough and capture those instances in a cluster. This can be done via automatic *horizon* adjustment. An adaptive sliding window technique like *ADWIN2* in [BG06] would be first candidate. If not in the algorithm itself, in MOA.

3) Another interesting area of improvement will be an automatic adjustment of the one parameter D_MAX, so that, apart from contracting/expanding the cluster, it could also contract/expand the area of the influence of the *leader* progressively in the multi-dimensional space. In this way, the algorithm would adjust itself automatically to *concept drift*, aiming at reaching an entirely autonomous stream clustering algorithm, needing no human intervention.

4) Expanding the use of *StreamLeader* to other domains would be also very important. A natural path for improvement would be heterogeneous data streams, where instances would be made up by numerical and categorical attributes. Here we could experiment with proximity measures, like *Gower* similarity coefficient. If MOA was expanded, *StreamLeader* could also theoretically handle textual domains, like document clustering, via calculation of similarity metrics like *TF-IDF* or other faster light-weight metrics. Using *StreamLeader* in Genetics or DNA analysis would be a potential area of work, where metrics like *Hamming* distances could cluster similar proteins, while probably requiring some sort of sketches to keep ordering of genes within the protein. Other options could be logical metrics like *Jaccard*, *Sokal* and so on.

5) A distributed version of *StreamLeader* would also be a natural development. We could integrate it into SAMOA (Scalable Advanced Massive Online Analysis). SAMOA is a distributed streaming machine learning framework that contains a programming abstraction for distributed machine learning algorithms. Because of its special design, number of *LeaderKernels* is always kept to a minimum and number of distance computations to *leaders* are therefore restricted. However, we should study whether substantial runtime improvement could be made or not with the use of distribution. We should bear in mind that the order in which the instances arrive does impact how *LeaderKernels* are created, but since *leaders* do evolve through re-calculation of all instances that fall within, then the ordering is not so critical in our algorithm. Therefore, parallelization of the data stream would be an option to consider.

5 Part 5 - Appendix

5.1 Stream clustering terminology

The following are high-level general definitions used in this work, intended to be used as quick reference:

Attribute: a quantity describing an instance. An attribute has a domain defined by the attribute type, which denotes the values that can be taken by an attribute.

Attribute (or variable-based) clustering: stream clustering model with the goal of finding attributes that behave similarly though time.

Batch mode: Batch machine learning techniques are used when one has access to the entire training data set at once.

Categorical: A finite number of discrete values. The type nominal denotes that there is no ordering between the values, such as last names and colors. The type ordinal denotes that there is an ordering, such as in an attribute taking on the values low, medium, or high.

Cluster feature vectors or *CF*: structure used for summarizing the data stream in the online phase by using an n-dimensional vector of numerical features that capture statistics representing a cluster.

Clustering algorithm: algorithm with the task of grouping a set of objects in such a way that objects in the same group (called a cluster) are more similar (in some sense or another) to each other than to those in other groups (clusters).

Concept drift: In streaming scenarios, change in underlying generation data distributions.

Coreset: small weighted set S of points, such that for any set of k-cluster centers, the weighted clustering cost of the coreset is an approximation for the clustering cost of the original set P with a smaller relative error.

Cosine similarity measure: measure of similarity between two vectors of an inner product space that measures the cosine of the angle between them.

Damped window model (or time fading models): time window model where more recent instances have higher importance than older ones by assigning normally weights to the instances so that more recent instances have higher weights than older ones, using for instance, decay functions.

Data set: A schema and a set of instances matching the schema.

Data stream: massive sequence of instances potentially unbounded.

Data stream mining: process of extracting knowledge structures from continuous, rapid data records forming a data stream.

Dimension: An attribute or several attributes that together describe a property. For example, a geographical dimension might consist of three attributes: country, state, city.

Distance: subgroup of proximity measures, where normally a numerical description is provided to describe how far apart objects are.

Euclidean distance: in Euclidean space, the distance between points x and y is the length $||x - y||_2$ of the straight line connecting them ($\overline{xy}$).

Feature vectors: an n-dimensional vector of numerical features that represent some object.

Grids: structures used for summarizing the data stream in the online phase by segmenting the d-dimensional feature space into equal parts and assigning density quantities to them.

Ground-truth (also true cluster): known cluster structure of a data set (or cluster labels).

Horizon: in a data stream, the set of instances to be considered whose weights are above a certain threshold.

Induction algorithm: An algorithm that takes as input specific instances and produces a model that generalizes beyond these instances.

Instance: A single object of the world from which a model will be learned, or on which a model will be used (e.g., for prediction). In most machine learning work, instances are described by feature vectors; some work uses more complex representations (e.g., containing relations between instances or between parts of instances).

Instance-based clustering: stream clustering model with the goal of finding instances that behave similarly though time.

Knowledge discovery: The non-trivial process of identifying valid, novel, potentially useful, and ultimately understandable patterns in data.

Landmark window model: time window model where portions of a data stream are separated by landmarks, which are relevant objects defined by, for example, time, or number of instances observed as velocity factor, producing non-overlapping sections of the stream and where instances arriving after a landmark are summarized until a new landmark appears.

LeaderKernel: extension of cluster feature vectors with expansion and contraction capabilities that allow to represent a potential true cluster in a streaming environment.

Machine learning: field of scientific study that concentrates on induction algorithms and on other algorithms that can be said to *learn*, and is one step in the knowledge discovery process.

Micro-cluster: see *cluster feature vector*.

Noise: recognized amounts of unexplained variation in a data sample.

Normalized space: space where *unit vectors* (spatial vectors of length 1) are chosen to form the basis of the space.

Object: see *instance*.

Object-based clustering: see *Instance-based clustering*.

Offline Phase: second phase of a stream clustering algorithm where final clustering is provided based on the components provided in the online phase.

Online (or data abstraction phase): second phase of a stream clustering algorithm where streaming data is summarized at a high level using special data structures.

Outlier: observation falling outside the overall pattern of a distribution.

Overfitting: effect occurred when a statistical model describes random error or noise instead of the underlying relationship.

Prototype arrays: Method used for abstracting the data stream in the online phase by summarizing

an array of representatives, where each one of them represents certain parts of the stream.

Proximity measure: metric that reflects the actual proximity between instances according to the final aim of the clustering, like distance, similarity or dissimilarity indexes.

Random RBF Generator: process that generates a random radial basis function stream. This generator was devised to offer an alternate concept type that is not necessarily as easy to capture with a decision tree model. The RBF (Radial Basis Function) generator works as follows: A fixed number of random centroids are generated. Each center has a random position, a single standard deviation, class label and weight. New examples are generated by selecting a center at random, taking weights into consideration so that centers with higher weight are more likely to be chosen. A random direction is chosen to offset the attribute values from the central point. The length of the displacement is randomly drawn from a Gaussian distribution with standard deviation determined by the chosen centroid. The chosen centroid also determines the class label of the example. This effectively creates a normally distributed hypersphere of examples surrounding each central point with varying densities. Only numeric attributes are generated.

Similarity index: subgroup of proximity measures as real-valued function that quantifies the similarity between two objects, taken in some sense as the inverse of distance metrics.

Sliding window: time window model where assumption is taken that recent data is more relevant than older data by using FIFO queue concept.

Stream clustering algorithm: clustering algorithm designed to work with data streams.

Time window model: model designed for data stream scenarios to detect potential change in data distribution generation processes by analyzing only most recent data.

True clustering (or ground-truth): see *ground-truth.*

Supervised learning: techniques used to learn the relationship between independent attributes and a designated dependent attribute (the label). Most induction algorithms fall into the supervised learning category.

Unsupervised learning: Learning techniques that group instances without a pre-specified dependent attribute. Clustering algorithms are usually unsupervised.

Weight (of an *instance*): in streaming scenarios using time window models, the numerical value assigned to an instance that decays as time elapses, normally represented as decay functions in damped window models.

Weight (of a *LeaderKernel*): The terms *weight* and *mass* are equal in this context and refer to the number of instances allocated to a *LeaderKernel.*

Window model: see *time window model.*

5.2 References

References

[Agg07] Aggarwal, C. 2007. Data Streams - Models and Algorithms. Springer.

[Agg09] C. Aggarwal. A Framework for Clustering Massive-Domain Data Streams. In ICDE Conference, 2009.

[AHWY03] Aggarwal, C. C., Han, J., Wang, J., and Yu, P. S. 2003. A framework for clustering evolving data streams. In Proceedings of the 29th Conference on Very Large Data Bases. 81-92.

[AHW+04] C. C. Aggarwal, J. Han, J. Wang, and P. S. Yu. A framework for projected clustering of high dimensional data streams. In VLDB, pages 852–863, 2004.

[ALY+10] C. Aggarwal, Y. Li, P. Yu, and R. Jin. On Dense Pattern Mining in Graph Streams, VLDB Conference, 2010.

[AMR+12] Ackermann, M. R., Martens, M., Raupach, C., Swierkot, K., Lammersen, C., and Sohler, C. 2012. Streamkm++: A clustering algorithm for data streams. ACM Journal of Experimental Algorithmics 17, 1.

[AV06] D. Arthur and S. Vassilvitskii. K-means++: The advantages of careful seeding. In Bay Area Theory Symposium, BATS 06, 2006.

[AV07] D. Arthur and S. Vassilvitskii. k-means++: the advantages of careful seeding. Proc. 18th ACM-SIAM Sympos. Discrete Algorithms, pp. 1027-1035, 2007.

[AY06] C. Aggarwal, and P. Yu. A Framework for Clustering Massive Text and Categorical Data Streams. In SIAM Conference on Data Mining, 2006.

[AY10] C. Aggarwal, and P. Yu. On Clustering Graph Streams, SDM Conference, 2010.

[AZY11] C. Aggarwal, Y. Zhao, and P. Yu. Outlier Detection in Graph Streams, ICDE Conference, 2011.

[B02] Barbara, D. 2002. Requirements for clustering data streams. SIGKDD Explorations (Special Issue on Online, Interactive, and Anytime Data Mining) 3, 23-27.

[BBD+02] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In Proc. 21st ACM Symposium on Principles of Database Systems, 2002.

[BC00] Barbará D, Chen P (2000) Using the fractal dimension to cluster datasets, in Proceedings of the Sixth ACM-SIGKDD International Conference on Knowledge Discovery and Data Mining, pages 260–264. ACM Press.

[BF07] W. Barbakh and C. Fyfe. Inverse weighted clustering algorithm. Computing and Information Systems, 11(2), 10–18, May 2007. ISSN 1352-9404.

[BF14] Bolanos M, Forrest J, Hahsler M (2014). "Clustering Large Datasets using Data Stream Clustering Techniques." In M Spiliopoulou, L Schmidt-Thieme, R Janning (eds.), Data Analysis, machine learning and Knowledge Discovery, Studies in Classification, Data Analysis, and Knowledge Organization, pp. 135–143. Springer-Verlag.

[BG06] A. Bifet and R. Gavalda. Learning from time-changing data with adaptive windowing. 2006. Poster. In 2007 SIAM International Conference on Data Mining (SDM'07), Minneapolis, Minnesota.

[BG09] A. Bifet and R. Gavalda. Adaptive Parameter-free Learning from Evolving Data Streams. 2009. IDA '09 Proceedings of the 8th International Symposium on Intelligent Data Analysis: Advances in Intelligent Data Analysis VIII, pp. 249-260

[BHK+11] A. Bifet, G. Holmes, R. Kirkby, and B. Pfahringer. 2011. DATA STREAM MINING: A Practical Approach. Tech. rep. University of Waikato. Retrieved from http://heanet.dl.sourceforge.net/project/moa-datastream/documentation/StreamMining.pdf.

[BHP+10] Bifet A, Holmes G, Pfahringer B, et al. MoA: massive online analysis, a framework for stream classification and clustering. Journal of machine learning Research. 2010;22:44–50.

[BK12] A. Bifet, R. Kirkby, Tutorial 1. Introduction to MOA Massive Online Analysis, March 2012. Retrieved from http://sourceforge.net/projects/moa-datastream/files/documentation/Tutorial1.pdf/download.

[BK+12] A. Bifet, R. Kirkby. Tutorial 2. Introduction to the API of MOA Online Analysis, March 2012. Retrieved from http://sourceforge.net/projects/moa-datastream/files/documentation/Tutorial2.pdf/download

[BSH+07] M. Brun, C. Sima, J. Hua, J. Lowey, B. Carroll, E. Suh, and E. R. Dougherty. Model-based evaluation of clustering validation measures. Pattern Recognition, 40(3),807–824, 2007.

[CEQZ06] Cao, F., Ester, M., Qian, W., and Zhou, A. 2006. Density-based clustering over an evolving data stream with noise. In Proceedings of the Sixth SIAM International Conference on DataMining. SIAM, 328-339.

[CM05] G. Cormode and S. Muthukrishnan. An Improved Data-Stream Summary: The Count-min Sketch and its Applications. In Journal of Algorithms, 55(1), 2005.

[CT06] T. Cover and J. Thomas. Elements of Information Theory (2nd Edition). Wiley-Interscience, 2006.

[CT07] Chen, Y. and Tu, L. 2007. Density-based clustering for real-time stream data. In KDD '07: Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining. ACM Press, 133-142.

[DH01] Domingos, P. and Hulten, G. 2001. A general method for scaling up machine learning algorithms and its application to clustering. In Proceedings of the 8th International Conference on machine learning. Morgan Kaufmann, 106-113.

[Don00] S. van Dongen. Performance criteria for graph clustering and markov cluster experiments. Technical Report INS-R0012, National Research Institute for Mathematics and Computer Science in the Netherlands, Amsterdam, May 2000.

[Dunn74] J. Dunn. Well separated clusters and optimal fuzzy partitions. Journal of Cybernetics, 4:95–104, 1974.

[EKS+96] Ester, M., Kriegel, H.-P., Sander, J., and Xu, X. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise. In 2nd International Conference on Knowledge Discovery and Data Mining. 226-231.

[EKW12] A. Eldawy, R. Khandekar, and K. L.Wu. On clustering Streaming Graphs, ICDCS Conference, 2012.

[ELL+10] B. Everitt, S.Landau, M. Leese, D. Stahl. Cluster Analysis, 5th Edition. Wiley, 2010.

[FC83] E. Folkes and C. Mallows. A method for comparingtwo hierarchical clusterings. Journal of the American Statistical Association, 78:553–569, 1983.

[Fis87] Fisher, Douglas H. (1987). "Knowledge acquisition via incremental conceptual clustering". machine learning 2 (2), 139–172

[FTT03] G. Flake, R. Tarjan, and M. Tsioutsiouliklis, Graph Clustering and Minimum Cut Trees, Internet Mathematics (2003), 1(4), pp. 385–408.

[Gam07] Gama, J. 2010. Knowledge Discovery from Data Streams. Chapman Hall/CRC.

[GG07] Gama, J. and Gaber, M. M. 2007. Learning from Data Streams: Processing Techniques in Sensor Networks. Springer.

[GK54] L.A. Goodman and W.H. Kruskal. Measures of association for cross classification. Journal of the American Statistical Association, 49:732–764, 1954.

[GMMO00] Guha, S., Mishra, N., Motwani, R., and O'Callaghan, L. 2000. Clustering data streams. In IEEE Symposium on Foundations of Computer Science. IEEE Computer Society, 359-366.

[GRL11] Gama, J., Rodrigues, P. P., and Lopes, L. 2011. Clustering distributed sensor data streams using local processing and reduced communication. Intelligent Data Analysis 15, 1, 3-28.

[Har75] J. Hartigan. Clustering Algorithms, chapter 3: Quick Partition Algorithms. Wiley, 1975.

[HCL+07] Q. He, K. Chang, E.-P. Lim, and J. Zhang. Bursty feature representation for clustering text streams. SDM Conference, 2007.

[HK01] J. Han and M. Kamber. Data Mining: Concepts and Techniques. Morgan Kaufmann, 2001.

[HXD+04] Z. He, X. Xu, S. Deng, and J. Huang. Clustering Categorical Data Streams, The Computing Research Repository, 2004.

[JZC06] Jain A, Zhang Z, Chang EY (2006) Adaptive non-linear clustering in data streams, CIKM, pp 122–131

[KABS11] Kranen, P., Assent, I., Baldauf, C., and Seidl, T. 2011. The clustree: indexing micro-clusters for anytime stream mining. Knowledge and Information Systems 29, 2, 249-272.

[KKJ+11] Kremer H, Kranen P, Jansen T, Seidl T, Bifet A, Holmes G, Pfahringer B (2011) An effective evaluation measure for clustering on evolving data streams. In: Proceedings of the 17thACMSIGKDD international conference on knowledge discovery and data mining, pp 868–876

[Kle02] J. Kleinberg, Bursty and hierarchical structure in streams, ACM KDD Conference, pp. 91–101, 2002.

[KR87] Kaufman, L. and Rousseeuw, P.J. (1987), Clustering by means of Medoids, in Statistical Data Analysis Based on the L_1–Norm and Related Methods, edited by Y. Dodge, North-Holland, 405–416.

[KR90] Kaufman, L. and Rousseeuw, P. 1990. Finding Groups in Data An Introduction to Cluster Analysis. Wiley Interscience.

[LL09] S. Lühr and M. Lazarescu (2009), "Incremental clustering of dynamic data streams using connectivity based representative points," Data and Knowledge Engineering, vol. 68, pp. 1-27.

[Maq67] MacQueen, J. B. 1967. Some Methods for Classication and Analysis of MultiVariate Observations. In 5th Berkeley Symposium on Mathematical Statistics and Probability, L. M. L. Cam and J. Neyman, Eds. Vol. 1. 281-297.

[Mei05] M. Meila. Comparing clusterings—an axiomatic view. InICML, pages 577–584, 2005.

[MM02] G. Manku, and R. Motwani. Approximate Frequency Counts over Data Streams, VLDB Conference,2002.

[MSE06] G. Moise, J. Sander, and M. Ester, "P3C: A Robust Projected Clustering Algorithm," Proc. Sixth IEEE Int'l Conf. Data Mining (ICDM), 2006.

[Ord03] C. Ordonez. Clustering Binary Data Streams with K-means. In Data Mining and Knowledge Discovery Workshop, 2003.

[PL07] Park, N. H. and Lee, W. S. 2007. Cell trees: An adaptive synopsis structure for clustering multi-dimensional on-line data streams. Data and Knowledge Engineering 63, 2, 528-549.

[Rand71] W.M. Rand. Objective criteria for the evaluation of clustering methods. Journal of the American Statistical Association, 66:846–850, 1971.

[RGP06] Rodrigues, P., Gama, J., and Pedroso, J. 2006. ODAC: Hierarchical clustering of time series data streams. In Proceedings of the Sixth SIAM International Conference on Data Mining. 499-503.

[RH07] A. Rosenberg and J. Hirschberg. V-measure: A conditional entropy-based external cluster evaluation measure. Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL), pages 410–420, 2007.

[Rij79] C.J.V. Rijsbergen. Information Retrieval (2nd Edition). Butterworths, London, 1979.

[RMJ07] M. Rattigan, M. Maier, and D. Jensen, Graph Clustering with Network Sructure Indices, ICML Conference Proceedings (2007), pp. 783–790.

[SKK00] Michael Steinbach, George Karypis, and Vipin Kumar. A comparison of document clustering techniques. In Workshop on Text Mining, KDD, 2000.

[Sha96] S. Sharma, Applied multivariate techniques. New York: Wiley, 1996.

[Str+13] F.Stahl. Tutorial 3. Introduction to MOA Clustering, October 2013. Retrieved from http://sourceforge.net/projects/moa-datastream/files/documentation/Tutorial3.pdf/download

[XW08] R. Xu, D. Wunsch. Clustering. Wiley, 2008.

[ZK04] Y. Zhao and G. Karypis. Empirical and theoretical comparisons of selected criterion functions for document clustering. ML, 55(3),311–331, 2004.

[ZRL97] Zhang, T., R. Ramakrishnon, y M. Livny. 1996. BIRCH: An efficient data clustering method for very large databases. Proceedings of the ACM SIGMOD Conference on Management of Data. Montreal (Canadá): ACM.

[ZS02] Zhu, Y. and Shasha, D. 2002. StatStream: statistical monitoring of thousands of data streams in real time. In Proceedings of the 28th International Conference on Very Large Data Bases. VLDB Endowment, 358-369.