Machine Learning (2021) 110:139–184
https://doi.org/10.1007/s10994-020-05905-4

Statistical hierarchical clustering algorithm for outlier detection in evolving data streams

Dalibor Krleža[1] · **Boris Vrdoljak**[1] · **Mario Brčić**[1]

Received: 16 September 2019 / Revised: 2 July 2020 / Accepted: 11 August 2020 /
Published online: 4 September 2020

Abstract
Anomaly detection is a hard data analysis process that requires constant creation and improvement of data analysis algorithms. Using traditional clustering algorithms to analyse data streams is impossible due to processing power and memory issues. To solve this, the traditional clustering algorithm complexity needed to be reduced, which led to the creation of sequential clustering algorithms. The usual approach is two-phase clustering, which uses *online* phase to relax data details and complexity, and *offline* phase to cluster concepts created in the *online* phase. Detecting anomalies in a data stream is usually solved in the *online* phase, as it requires unreduced data. Contrarily, producing good macro-clustering is done in the *offline* phase, which is the reason why two-phase clustering algorithms have difficulty being equally good in anomaly detection and macro-clustering. In this paper, we propose a statistical hierarchical clustering algorithm equally suitable for both detecting anomalies and macro-clustering. The proposed algorithm is single-phased and uses statistical inference on the input data stream, resulting in statistical distributions that are constantly updated. This makes the classification adaptable, allowing agglomeration of outliers into clusters, tracking population evolution, and to be used without knowing the expected number of clusters and outliers. The proposed algorithm was tested against typical clustering algorithms, including two-phase algorithms suitable for data stream analysis. A number of typical test cases were selected, to show the universality and qualities of the proposed clustering algorithm.

Keywords Big data · Clustering · Anomaly detection · Fraud detection

Editor: Joao Gama.

This research has been supported by the European Regional Development Fund under the Grant KK.01.1.1.01.0009 (DATACROSS).

✉ Dalibor Krleža
dalibor.krleza@fer.hr

Boris Vrdoljak
boris.vrdoljak@fer.hr

Mario Brčić
mario.brcic@fer.hr

[1] Faculty of Electrical Engineering and Computing, University of Zagreb, Unska 3, Zagreb, Croatia

1 Introduction

Today, we create, collect, and process more data than ever before. All this data holds many patterns of interest. Most of the patterns are regularly occurring in the data. Finding these typical patterns can help to identify outliers, i.e., anomalies that occur sparsely. The more data is generated, the more patterns and outliers we are able to find, which leads to the big data paradigm, i.e., endless data streams that need to be continuously analysed in search of typical data patterns and outliers.

Data clustering algorithms are one of many solutions that can be used to perform analysis of endless data streams. Using clustering algorithms to analyse data in an endless data stream encounters typical processing power and memory limitation issues. Besides these, there are additional data related considerations. For example, we never know how many clusters we can expect. A set of outliers could start forming a new cluster. As the data stream evolves, more and more clusters can be formed from outliers. Another difficulty is that the underlying data generation processes can be random and there is no guarantee of the order in which data are generated, which is different from the data processing order mentioned in Silva et al. (2013). This can significantly affect how clusters are formed and their evolution. Next, the underlying data generating processes can change over time (Gama 2010), generating data whose statistical distribution changes over time, which leads to drifting data clusters. This requires a change of the cluster statistical distribution, to remain aligned with the related data generating process. Cluster drifting can lead to cluster agglomeration or splitting, making clustering algorithms more complex.

There are many clustering algorithms that can process endless data streams and detect outliers. Our testing of these algorithms shows that clustering algorithms rarely perform equally good when detecting both data clusters and outliers. Either outlier detection works good and clusters become too fragmented, or clusters are detected correctly with outliers being included into clusters.

Historically, due to limited processing power and memory, we needed to reduce clustering algorithm complexity and find solutions that are capable of processing data streams. We cannot move back and forth in the processed data stream, randomly accessing data objects, which automatically eliminates clustering algorithms that have double iterations and $O(n^2)$ complexity, such as *k-means*. Most of the data stream clustering algorithms are taking the two-phase approach (Silva et al. 2013). Such an approach divides the clustering process into two phases. The first phase, named *online* phase, reduces the complexity of the retrieved data. The online phase consists of an algorithm that maps input data into a set of concepts named *micro-clusters*. This mapping tends to simplify calculations and memory needed to store micro-clusters. It seems that many algorithms have reduced universality and capabilities because of the overly simplistic online phase. While this is historically understandable, hardware processing power and memory are constantly advancing since the conception of a first data streaming clustering algorithm. The set of micro-clusters is then picked up by the second phase, named *offline* phase, and finally clustered into a set of *macro-clusters*. The offline phase algorithm does the mapping between micro and macro-clusters. These two phases can be part of the same algorithm or can be separate algorithms. In any of these cases, anomaly detection happens in the online phase, where full data details are available. The offline phase algorithm can sometimes reduce the accuracy of the anomaly detection given by the online phase, e.g., *k-means* in the offline phase will most certainly merge outliers

with nearby clusters. Algorithms intended for the online phase are creating simple geometrical, statistical, or density-based micro-clusters, which cannot be taken as the final clustering result.

In this paper, we propose *Statistical Hierarchical Clustering* (SHC) algorithm for outlier detection in evolving data streams. SHC is a **sequential clustering algorithm** suitable for use in data stream processing, i.e., it accesses and retrieves each data object from the input data stream only once and sequentially. After retrieving data object, SHC is performing a **single-phase clustering**, and returns clustering results immediately. The returned clustering results comprises a classified component or outlier (micro-cluster level) and cluster (macro-cluster level). SHC is capable of performing outlier detection, component population forming and updating, and clustering in the same step. This is enabled by the **statistical agglomeration concept**, which allows outliers and components to be agglomerated based on the statistical relations between them. Statistical agglomeration allows forming of new components from a set of outliers, assimilation of outliers by growing components, or merging two or more components under the same arbitrarily shaped cluster. This sequence of actions is the key for universality of SHC, allowing it to be equally good in outlier detection and clustering. Also, the statistical agglomeration allows SHC to work without supplying the expected number of clusters k in advance, which is an important feature for a data streaming clustering algorithm, since in an endless evolving data stream the number of clusters and outliers constantly changes. Statistical agglomeration does not come without issues. Outlier assimilation from the growing components seems to be one of the most processing power and time consuming tasks. To ease the number of checked outliers that can be assimilated, SHC components are keeping track of the **statistical neighbourhood**, allowing SHC to focus only on outliers and components that are within the statistical interaction range. In many cases, we want to know that a newly processed data object from the input data stream is part of a population that drifted away or got separated into multiple sub-populations from its initial formation. Such information is impossible to obtain if we do not allow micro-clusters to follow the population evolution or there is no mapping between micro and macro-clusters. With SHC, we propose population evolution tracking on the component level. This is achieved through a novel concept of **sub-clustering**. To analyse and capture the statistical change in the component population, each component uses additional child SHC instance to cluster the latest population members. The results of such component sub-clustering can be interpreted as population move or separation into several sub-populations, i.e., **component drift or split**. Such an approach allows **population traceability**, i.e., it allows two temporally distant data objects in the input data stream to be recognized as members of the same evolving population, which was a proposed research by Nguyen et al. (2015). On the cluster level, each component drift or split can result in cluster split, hence traceability is supported on the cluster level as well. All this is computationally more complex than current two-phase data stream clustering algorithms. One of the key questions this paper is going to answer whether it is possible to have computationally complex algorithm and still be able to process data stream with satisfactory speed.

We performed comparative testing and analysis between SHC and other similar clustering algorithms. The evaluation was done using synthetic and real-life datasets and data streams. Using synthetic data streams we were able to control execution and uniformity of testing for all tested clustering algorithms. We executed various testing scenarios to cover all SHC functionalities and compare them to other algorithms, which included static clustering, evolving data streams comprising drifting concepts, and outlier detection. The proposed SHC algorithm performed exceptionally well in most of these scenarios.

Many clustering indices (Desgraupes 2017) favor correct clustering instead of outlier detection. Due to the low outlier share in the overall data stream, outlier detection errors are not significantly affecting most of the clustering indices, such as the corrected Rand Index (CRI) (external), or purity (internal). In this paper we proposed a simple framework for outlier detection accuracy assessment, for which we used modified Jaccard Index (Desgraupes 2017; Jaccard 1912), hereby named *Outlier Jaccard Index* (OJI). For the purpose of data stream outlier detection assessment, we used a cumulative variant of OJI (cOJI), which counts all the true positive, false positive and undetected outliers across the whole tested data stream.

This paper is organized as follows: the next section gives an overview of the related work. In Sect. 3 we elaborate how multivariate normal distribution can be used for data stream clustering and outlier detection purposes. In Sect. 4 we give detailed definition and description of SHC. SHC is evaluated in Sect. 5. The conclusion is given in Sect. 6.

2 Related work

The most basic sequential clustering algorithms are ***sequential k-means clustering***, ***sequential agglomerative clustering***, and ***sequential nearest-neighbour clustering*** (Ackerman and Dasgupta 2014). These sequential clustering algorithms are giving poorer results than their batch variants for obvious reasons: limited processing power and memory for given high data volume and velocity are preventing us from optimally realigning clusters. Also, knowing the expected number of clusters k in advance is a drawback for using these algorithms in the data stream context.

The purpose of the online phase is to develop data structures that hold micro-cluster data and are used by the clustering algorithm in the offline phase. The most notable data structures produced in the online phase are cluster feature (CF) vectors, first introduced in ***BIRCH*** (Zhang et al. 1996). BIRCH represents an attempt to fight limited memory in data stream clustering. Data objects from data streams cannot be stored in memory, i.e., they need to be processed by the clustering algorithm and dropped. CFs are organized in a hierarchical tree following the hierarchical clustering principle (Jain et al. 1999) and are storing basic data about a captured micro-cluster, such as center, radius, and density. Later, in the offline phase, a batch clustering algorithm, such as hierarchical agglomerative clustering or *k-means* clustering, can be used on micro-clusters to produce the final clustering results or macro-clusters.

There are many extensions of BIRCH, such as ***BICO*** (Fichtenberger et al. 2013). While BIRCH decides heuristically how to merge points into micro-clusters residing in CF tree leaves, BICO refines this process using a k-means coreset. In BICO, each CF stores a representative as well. New points are added in the CF tree and merged with existing CFs if similar. This means that BICO can store outliers in the CF tree. BICO needs to know the expected number of clusters k in advance.

CluStream (Aggarwal et al. 2003) is another extension of BIRCH. CluStream micro-clusters are temporal based extension of the CF concept. It allows storing a micro-cluster statistic snapshot at a particular time frame in the input data stream. Additionally, a pyramidal time frame concept is used to store micro-cluster snapshots at different levels of time frame granularity. This allows CluStream in the online phase to build a set of micro-clusters, including outliers, capturing temporal and spatial statistics on a different level of temporal granularity.

DenStream (Cao et al. 2006) uses the same two-phase approach and the CF tree as in BIRCH. In the online phase, DenStream creates two types of CFs, *core* and *outlier* micro-clusters. Core micro-clusters have their weight above the defined weight threshold, while outlier micro-clusters have their weight below the defined threshold. A time decay function is applied to each CF, so micro-clusters make less impact over time. A new data object is added to the closest core or outlier micro-cluster only if this addition does not increase the radius of the targeted micro-cluster above the radius threshold. If both addition attempts fail, the new data object creates a new outlier micro-cluster. Once the weight of an outlier micro-cluster gets above the weight threshold, it is promoted to a potential core micro-cluster. The offline phase in DenStream is done with a variant of the DBSCAN algorithm (Ester et al. 1996).

ClusTree (Kranen et al. 2011) is another extension of BIRCH. In the online phase, it uses the CF tree as an indexing tree. The closest CF leaf is reached descending down the tree by checking which branch is closer to the newest data object. When the closest CF leaf is reached, the processed data object is inserted as a child CF. The indexing CF tree maintained by ClusTree is regularly reorganized when a CF node gets full. Each CF node has an additional buffer, capable of holding data objects. This way, when the input data stream delivers data objects faster than ClusTree is capable of processing, data objects keep piling up in the CF tree buffers, delaying their processing (descending) for a later time, when the input data stream slows down.

Another density based method is ***D-Stream*** (Chen and Tu 2007). D-Stream uses a grid placed in the attribute space Ω. In the online phase, D-Stream only calculates density in the grid cells. This density decays over time. Sparse cells get removed from the grid after a while. In the offline phase, neighbour dense cells are grouped in macro-clusters.

DBSTREAM (Hahsler and Bolaños 2016) is similar to D-Stream. The online phase in DBSTREAM updates micro-clusters, which are defined by its center, radius, and density. A new data object can update multiple micro-clusters, those in which it falls within a radius from the center. If there are micro-clusters that can absorb new data object, their centers and densities get updated. If the new data object does not fall within any of the already existing micro-clusters, a new micro-cluster is created. Since micro-clusters can overlap, they can share some similar density, which is the property used in the offline phase to create macro-clusters. DBSTREAM creates a graph that connects overlapping micro-clusters sharing similar density, which can result in a set of disjunct sub-graphs. Each sub-graph represents one macro-cluster. Similar mechanism is used in the statistical agglomeration of SHC. Although data objects can update micro-cluster position, its drifting and moving is limited. Instead, DBSTREAM relies on the shared density graphs to resolve population drift and separation.

In the k nearest neighbours (kNN) method, a potential outlier can be determined by limiting distance needed to reach k nearest neighbours from the new data object. However, knowing k nearest neighbours means knowing previous data objects obtained from the input data stream. The fact that the number of outliers constitutes a small portion of all data is used for intelligent windowing and temporal data fading in ***kNN LEAP*** (Cao et al. 2014), which can be applied to data streams.

ScaleKM is a scalable extension of *k-means* clustering algorithm (Bradley et al. 1998). Although originally intended for processing of large databases, this algorithm can be used for data streams as well. Clusters are represented by the CF tree as in BIRCH. In the online phase, a compression step is performed that uses the CF tree and Mahalanobis distance (De Maesschalck et al. 2000) to discard new data objects that statistically belong to one of the known clusters. This is different from all previous distance-based methods, as it uses the multivariate

normal distribution to model clusters. All remaining data objects are then passed to the offline phase, where *k-means* clustering is applied. Clusters created in the offline phase (their CFs) are then merged among themselves and with existing clusters from the online phase using hierarchical agglomerative clustering (Rasmussen 1992).

Online Elliptical Clustering (OEC) (Moshtaghi et al. 2016) is a statistical algorithm that, similarly to ScaleKM, uses the multivariate normal distribution to form clusters from the input data stream. OEC adds a *guard-zone* to each cluster, i.e, a statistical area around highly-populated clusters to protect them from random noise (random outliers) in the surrounding area. New clusters are formed using a *state tracker* that absorbs new outliers and uses the *c-separation* measure (Dasgupta 1999) to establish a new cluster.

Another way of detecting an outlier is through comparing data sets histograms (Solaimani et al. 2014). Using the windowing method, similar to **ADWIN** (Bifet and Gavalda 2007), we can select a subset of a data stream and calculate its histogram. By knowing that there are no outliers in the captured window, we can collect histograms of windows where there were no outliers detected. An outlier is found by detecting a change in histograms of two distinctive windows using the chi-square method (Press et al. 2007), knowing that one of the windows comprised no outliers. Authors in Solaimani et al. (2014) focus on an interesting modern big data analytic platform..

Continuous Outlier Detection (COD) (Kontaki et al. 2016) and its variants are a set of clustering algorithms focused on outlier detection in data streams. COD uses a sliding window concept for capturing and detecting outliers from the input stream. For newly retrieved data object p from the input data stream, a neighbourhood set of data objects P_p is queried from the sliding window data objects. All data objects that are closer than R from the new data object p are considered to be part of P_p. If this set is smaller than k, the newly retrieved object p is considered for an outlier and placed in the set of outliers $\mathscr{D}(R, k)$. Additionally, COD keeps track of preceding and succeeding neighbours. **Micro-cluster-based Continuous Outlier Detection** (MCOD) variant additionally uses a set of statical $R/2$ wide hyperspherical micro-clusters, each having at least $k + 1$ members. If the new data object p is not within any of the established micro-clusters, then it is a candidate for an outlier. All these algorithms keep an event queue, which is regularly processed and is a specific implementation of the fading mechanism found in other clustering algorithms. COD algorithms are capable of turning inliers into outliers due to the event queue mechanism. We do not agree about retroactive redeclaring of an inlier into an outlier.

3 Statistics in data stream clustering

SHC classifies the input data into two types of objects: outliers and components made of data populations that fit multivariate normal distribution (Gut 2009). Clusters, which are formed from components, are arbitrarily shaped through a mixture of their component statistical distributions (Dasgupta 1999). Component membership is determined through Mahalanobis statistical distance (De Maesschalck et al. 2000). A data object that statistically does not fit any component is considered to be an outlier.

A **data stream** is defined as an endless sequence of data objects (Aggarwal 2007; Gama and Gaber 2007; Gama 2010; Silva et al. 2013).

$$\begin{aligned} \mathscr{D}(t) &= \{d_1, d_2, ..., d_m\} \\ \lim_{t \to \infty} |\mathscr{D}(t)| &= \infty \end{aligned} \tag{1}$$

Every data object in the data stream is a vector of values (attributes) that can belong to the same n-dimensional attribute space Ω_n. If a data stream is composed of different data sources, then data objects can belong to different attribute spaces.

When saying that a data stream is evolving, we refer to the statistical change of data in the processed data stream. In the beginning, all we have is a set of outliers. After a while, some outliers start grouping, i.e., forming populations of related data objects. Populations can disappear after some time, drift away from the starting position, or get divided into multiple disconnected populations. All this represents the evolution of data in the processed data stream. Depending on the context and the underlying data, there are many ways of modelling a data population. The most common research findings suggest that natural processes produce normally distributed data. Is normality in such data something to be assumed? Some researchers (Micceri 1989; O'Boyle Jr and Aguinis 2012) suggest that the natural processes produce contaminated data, which can be only modelled through a mixture of distributions, and that not all natural processes adhere to the mathematical beauty of the normal distribution. SHC uses normal multivariate models (Gut 2009) for modelling data populations on the component level. Non-normally distributed data can be modelled through a mixture of multivariate normal distributions (Baudry et al. 2010). In SHC, a cluster is a set of components, which allows the cluster to be a mixture of multivariate normal distributions.

3.1 Statistical component

Let us define a **population** of data objects $\mathbb{X}(t_1) \subseteq \mathscr{D}(t_1)$ in the attribute space Ω_n such that $|\mathbb{X}(t_1)| = m$. Each data object in the population can be expressed as $X^k = [{x_1}^k, {x_2}^k, ..., {x_n}^k]^\top \in \mathbb{X}(t_1)$. A multivariate normal distribution over the population is defined as

$$X^k \sim \mathscr{N}_n^m(\mu^m, \Sigma^m) \tag{2}$$

where $\mu^m = E[X^k] = [E({x_1}^k), E({x_2}^k), ..., E({x_n}^k)]^\top$ is the mean vector, and $\Sigma^m = E[(X^k - \mu^m)(X^k - \mu^m)^\top]$ is the covariance matrix of the population $\mathbb{X}(t_1)$. The covariance matrix is

$$\begin{aligned}\Sigma^m &= \begin{bmatrix} {\sigma_1}^2 & {\sigma_{(1,2)}}^2 & \cdots & {\sigma_{(1,n)}}^2 \\ {\sigma_{(1,2)}}^2 & {\sigma_2}^2 & \cdots & {\sigma_{(2,n)}}^2 \\ \cdots & \cdots & \cdots & \cdots \\ {\sigma_{(1,n)}}^2 & {\sigma_{(2,n)}}^2 & \cdots & {\sigma_n}^2 \end{bmatrix} \\ &= \begin{bmatrix} {\sigma_1}^2 & \rho_{(1,2)}\sigma_1\sigma_2 & \cdots & \rho_{(1,n)}\sigma_1\sigma_n \\ \rho_{(1,2)}\sigma_1\sigma_2 & {\sigma_2}^2 & \cdots & \rho_{(2,n)}\sigma_2\sigma_n \\ \cdots & \cdots & \cdots & \cdots \\ \rho_{(1,n)}\sigma_1\sigma_n & \rho_{(2,n)}\sigma_2\sigma_n & \cdots & {\sigma_n}^2 \end{bmatrix}\end{aligned} \tag{3}$$

where ${\sigma_{(i,j)}}^2$ represents covariances between dimensions i and j for $\forall i, j \in [1, n] \mid i \neq j$. Each covariance can be expressed as ${\sigma_{(i,j)}}^2 = \rho_{(i,j)}\sigma_i\sigma_j$, a product of two dimensional standard deviations and a correlation factor $\rho_{(i,j)}$, where $\rho_{(i,j)} = \rho_{(j,i)}$.

A new data object $X^{m+1} \in \mathbb{X}(t_2) \subseteq \mathscr{D}(t_2) | t_2 > t_1$ added to this population must update distribution to $\mathscr{N}_n^{m+1}(\mu^{m+1}, \Sigma^{m+1})$ using some incremental technique, such as Welford's online variance update (Welford 1962).

$$\begin{aligned} \mu^{m+1} &= \mu^m + \frac{X^{m+1} - \mu^m}{m+1} \\ \Sigma^{m+1} &= \Sigma^m + \frac{(X^{m+1} - \mu^m)(X^{m+1} - \mu^{m+1})^\top - \Sigma^m}{m+1} \end{aligned} \tag{4}$$

Previously defined population distribution is used to define a component in SHC.

3.2 Statistical distance

In the statistical distribution $\mathcal{N}_n(\mu, \Sigma)$, a data object $X \in \mathbb{X}(t)$ is $[X - \mu]$ distant from the mean value of the distribution. Mahalanobis statistical distance (De Maesschalck et al. 2000) is defined as

$$d_\sigma(X, \mu, \Sigma) = \sqrt{[X - \mu]^\top \Sigma^{-1} [X - \mu]} \tag{5}$$

The main issue in using the statistical distance in the clustering algorithms is its calculation expense and volatility. Calculating an inverse of a covariance matrix is an expensive operation. This calculation needs to be done for each data object multiplied by the number of components and outliers, which needs high processing power and is the main reason why many researchers turn to density-based clustering methods. Another issue is when a covariance matrix becomes singular for which we cannot calculate its inverse, i.e., only pseudo-inverse can be calculated. This mainly happens in components that have low population, and when a dimension in the component population has no variance, i.e., $\exists i \in [1, n] : {\sigma_i}^2 = 0$. In the case of singularity, a replacement covariance matrix is defined as

$$\Sigma' = diag(\Sigma) \tag{6}$$

where all missing dimension variances are replaced with virtual variance ${\sigma_v}^2$. For example

$$\Sigma = \begin{bmatrix} 0.3 & 0.2 & 0 \\ 0.2 & 1.3 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \Sigma' = \begin{bmatrix} 0.3 & 0 & 0 \\ 0 & 1.3 & 0 \\ 0 & 0 & {\sigma_v}^2 \end{bmatrix} \tag{7}$$

We preserve the original covariance matrix Σ, as each new data object retrieved from the processed data stream can update the matrix to become non-singular. Once covariance matrix becomes non-singular, instead of performing (4), we can invert the covariance matrix and then do incremental updates (Nguyen et al. 2015) using Sherman–Morrison–Woodbury formula (Sherman and Morrison 1950; Woodbury 1950). A similar update is done for OEC (Moshtaghi et al. 2016).

$$\begin{aligned} u &= \frac{X^{m+1} - \mu^m}{\sqrt{m}}, v = \frac{X^{m+1} - \mu^{m+1}}{\sqrt{m}} \\ (\Sigma^{m+1})^{-1} &= \frac{m+1}{m}((\Sigma^m)^{-1} - \frac{(\Sigma^m)^{-1} u v^\top (\Sigma^m)^{-1}}{1 + v^\top (\Sigma^m)^{-1} u}) \end{aligned} \tag{8}$$

So obtained inverse of the covariance matrix can be used directly in calculating statistical distance (5). Though (8) seems to be computationally more demanding than (4), avoiding to calculate the matrix inverse for each and every statistical distance calculation is

beneficial as the number of dimensions n, components, and outliers grow. For matrix inversion using Cholesky decomposition, the computational complexity is $O(n^3)$ (Krishnamoorthy and Menon 2013). Thus, justification of trading the time needed for the distribution update to get faster statistical distance calculation becomes visible in high-dimensional and large data sets, i.e., endless data streams. We consider data object X^{m+1} to be a part of a component with distribution $\mathcal{N}_n^m(\mu^m, \Sigma^m)$ only if its statistical distance is less than a threshold θ. Membership-wise, we call our components θ-**bound components**, since the membership for the component distribution is cut at the statistical distance θ.

$$d_\sigma(X^{m+1}, \mu^m, \Sigma^m) \leq \theta \tag{9}$$

3.3 Outlier detection

According to the multivariate normal distribution probability density function, there is a small probability that a data object acquired from a data stream can be a member of any component. Since SHC is a crisp algorithm, i.e., we are not considering fuzzy clustering, to define what is an outlier we need θ-bound components as defined in (9). Based on that, we can precisely define that a data instance is an outlier if it is not a member of any θ-bound component.

An outlier, being a single data object, has no statistic distribution we can use to model it. To keep statistic distance (5) calculations consistent, we construct a virtual statistic distribution around the outlier, the same way as for the missing dimensions in (7). Here, the virtual variance ${\sigma_v}^2$ can be fixed or can be calculated from the outlier surroundings, e.g., derived from the closest component covariance matrix. The outlier virtual covariance matrix is calculated as

$$\Sigma_v = {\sigma_v}^2 I_n \tag{10}$$

Outlier o can be modelled as a virtual multivariate normal distribution

$$\mathcal{N}_n(o, \Sigma_v) \tag{11}$$

3.4 Variance limitation

In certain cases we want to limit outlier and component updates. We define a variance limitation function for the covariance matrix Σ as

$$\theta_{lim}(\Sigma, {\sigma_\theta}^2) = \begin{cases} 0, & \nexists {\sigma_{(i,j)}}^2 \in \Sigma | {\sigma_{(i,j)}}^2 > {\sigma_\theta}^2 \\ 1, & \text{otherwise} \end{cases} \tag{12}$$

resulting in 1 when the variance limit ${\sigma_\theta}^2$ has been reached, and 0 when the variance limit has NOT been reached.

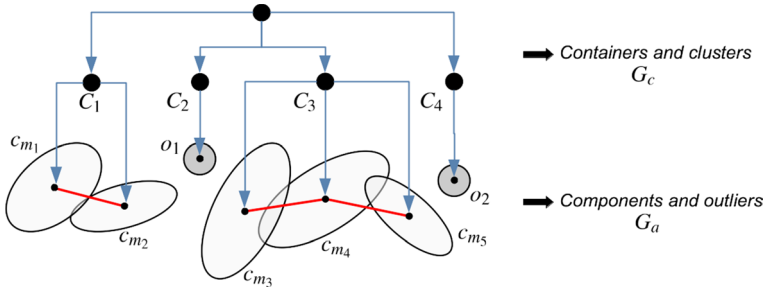


Fig. 1 SHC supporting structures

4 The SHC algorithm

4.1 Supporting structures and classification objects

Similar to other algorithms, SHC maintains structures that support organization and manipulation of main *classification objects* (co), namely outliers and components. SHC supporting structures can be seen in Fig. 1. SHC uses two distinct graphs, one vertical and one horizontal.

The vertical graph G_c serves as a *container tree* and supports grouping of the classification objects. We define G_c as an acyclic and directed graph where each node can have one or more child nodes. Edges in G_c are directed from parent to child nodes.

$$\begin{aligned} G_c &= (N_c, E_c), E_c = N_c \times N_c \\ N_c &= T \cup Cl \cup Cm \cup O \end{aligned} \tag{13}$$

We divide container tree nodes into containers T, clusters Cl, components Cm and outliers O, where $Co = Cm \cup O$ are classification objects. Each leaf node in G_c is a classification object.

$$\begin{aligned} &\nexists (n_s, n_t) \in E_c : n_s \in Co \\ &\forall n_t \in Co \exists (n_s, n_t) \in E_c : n_s \in Cl \end{aligned} \tag{14}$$

By traversing G_c from a classification object up to the root node, we can find a node that represents the cluster of the starting classification object. A cluster node $C_i \in Cl$ is a node in G_c that has only leafs, as defined in (14) and seen in Fig. 1. All other nodes above clusters are containers that define the structure.

$$\forall n_i \in T \exists n_j \in (T \cup Cl) : (n_i, n_j) \in E_c \tag{15}$$

The horizontal graph G_a serves as an *agglomeration graph* and supports the statistical agglomeration mechanism in SHC. Nodes of G_a can only be components. This graph is similar to the shared-density graph in DBSTREAM. In case of SHC, there is no density measure. Instead, edges of G_a store shared population between two population-sharing components. We define the agglomeration graph as a cyclic undirected graph

$$\begin{aligned} G_a &= (N_a, E_a), N_a \subseteq Cm, E_a = N_a \times N_a \times \mathbb{N} \\ cc_{(s,t)} &= (cm_s, cm_t, sp_{(s,t)}) \in E_a \end{aligned} \tag{16}$$

where $sp_{(s,t)} \in \mathbb{N}$ is the number of shared population members between components cm_s and cm_t. It is important to notice that some components do not need to be included in the agglomeration graph G_a, which means they are not currently connected to any other component.

Definition 1 Component A component is a tuple derived from its modelling multivariate normal distribution $\mathcal{N}_n(\mu, \Sigma)$

$$cm_i = (\mu_i, \Sigma_i, p_i, N_i, cb_i, SHC_i, R_i, \delta_i, \phi_i) \in Cm \tag{17}$$

where μ_i is the mean of the component distribution, Σ_i is the covariance matrix, p_i is the component population number, N_i is the statistical neighbourhood of the component, cb_i is the component baseline and SHC_i is the child SHC used for analysing of the component drift and split, R_i is the redirection component, δ_i is the component decay counter, and ϕ_i are component flags. The component is θ-bound (9), meaning that all data object statistically closer to the mean μ_i than θ, are considered to be members of the component population.

Definition 2 Outlier An outlier is a tuple derived from its modelling virtual multivariate normal distribution $\mathcal{N}_n(X^k, \Sigma_v)$

$$o_i = (\mu_i = X^k, \Sigma_i = \Sigma_v, p_i = 1, N_i = \emptyset, cb_i = \varnothing, SHC_i = \varnothing, R_i = \varnothing, \delta_i, \phi_i) \in O \tag{18}$$

where μ_i is the outlier value, i.e., data object X^k, Σ_i is the virtual covariance matrix, δ_i is the outlier decay counter, and ϕ_i are outlier flags. All the other elements in the tuple are not applicable to outliers. Tuple similarity between components and outliers is obvious and intentional, to allow consistent algebraic calculations for SHC.

4.2 Statistical agglomeration

4.2.1 Classification

For each new data object X^{m+1} retrieved from the input data stream, we query classification objects to find those that contains the new data object within the θ statistical distance, i.e., within their populations as defined in (9). Our statistical distance query results with two sets

$$\begin{aligned} &Q_c, Q_n \subseteq Co \\ &\forall co_i \in Q_c : d_\sigma(X^{m+1}, \mu(co_i), \Sigma(co_i)) \leq \theta \\ &\forall co_j \in Q_n : \theta < d_\sigma(X^{m+1}, \mu(co_j), \Sigma(co_j)) \leq p_n\theta \end{aligned} \tag{19}$$

where Q_c is the set of classification objects having the new data object X^{m+1} within the statistical distance θ, named *classified set*, and Q_n is the set of classification objects in the neighbourhood that do not directly comprise X^{m+1} in their populations, but are close enough for some future interactions, named *neighbourhood set*. Classification objects in Q_c are named *classified objects* as the input data object X^{m+1} can be classified to any of them.

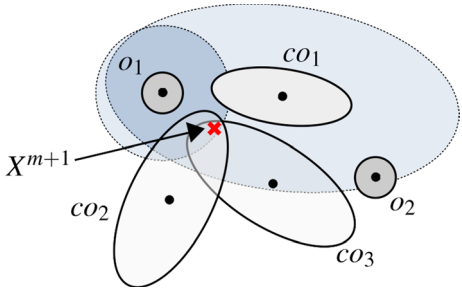


Fig. 2 Query example

$p_n > 1$ represents an SHC parameter that defines the maximal statistical neighbourhood distance.

The example in Fig. 2 shows new data object X^{m+1} being classified. With $p_n = 3$, the result of the statistical distance query is $Q_c = \{co_2, co_3\}, Q_n = \{co_1, o_1\}$.

In case we get an empty classified set $Q_c = \emptyset$, we consider data object X^{m+1} for an outlier. In this case we need to create a new outlier descriptor (11), add it to the set O and container tree G_c. If we get exactly one component in the classified set $|Q_c| = 1$, this can be either a component or an outlier, which can be directly updated as defined in Sect. 3. For an outlier, this means promotion to a component when (12) is satisfied. This way we want to prevent that repetitive or close data objects promoting outliers to small components.

If we get more classified objects in the classified set $|Q_c| > 1$, this means that data object X^{m+1} is a shared member in the Q_c populations. In such case we look for the minimal statistical distance in the classified set

$$co_c = \underset{co_i \in Q_c}{arg\ min}\ d_\sigma(X^{m+1}, \mu(co_i), \Sigma(co_i)) \tag{20}$$

For all outliers in the classified set $O \cap Q_c$ this means forming a single component with co_c, no matter if co_c was originally a component or an outlier. For all other components it means potential agglomeration under the same cluster.

4.2.2 Shared population agglomeration

Having $|Q_c| > 1$ means that we have more classified objects that share the new data object X^{m+1}. As already mentioned, all outliers in the classified set $O \cap Q_c$ are merged into one resulting component co_c. After this, we assume that Q_c comprises only existing components $Q_c \subseteq Cm$, i.e., X^{m+1} is truly a shared member of multiple populations. Agglomeration of such remaining components in Q_c is resolved by the agglomeration graph G_a. Each pair of components in Q_c must have an edge in G_a

$$\forall co_i, co_j \in Q_c \exists (co_i, co_j, sp_{(i,j)}) \in E_a : co_i \neq co_j \wedge sp_{(i,j)} \geq 1 \tag{21}$$

where $sp_{(i,j)}$ is the total number of a shared population members previously encountered between components co_i and co_j. This is consistent with G_a definition (16) where nodes of the agglomeration graph can be comprised only of components. Here we introduce a SHC parameter θ_{sp} which is a required minimal number of shared population members between two components for their agglomeration under the same cluster. The final agglomeration under the same cluster is done in the *container tree* G_c as seen in Fig. 3. Therefore, each pair of components under the same cluster $C_i \in Cl$ in the container tree G_c must satisfy

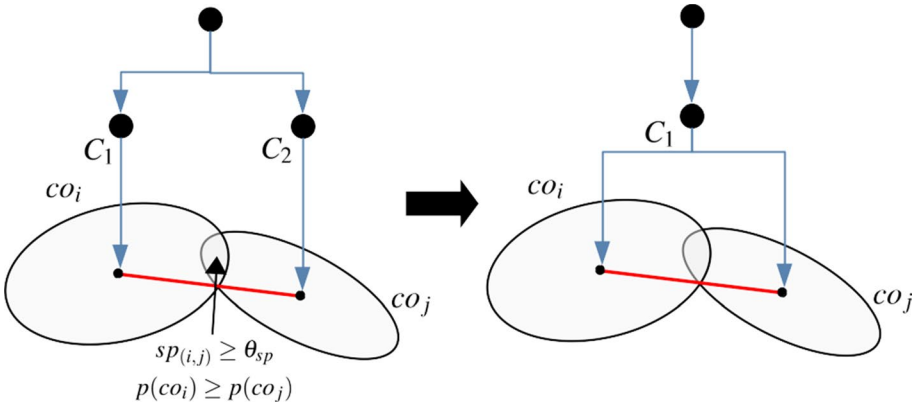


Fig. 3 Shared population member agglomeration example

$$\exists C_i \in Cl, \exists co_i, co_j \in Cm : (co_i \neq co_j \wedge (C_i, co_i) \in E_c \wedge (C_i, co_j) \in E_c \wedge \exists (co_i, co_j, sp_{(i,j)}) \in E_a : sp_{(i,j)} \geq \theta_{sp}) \tag{22}$$

When agglomerating two components under the same cluster, we retain the cluster that has more population members

$$p(C_i \in Cl) = \sum_{\forall co_j \in Cm : (C_i, co_j) \in E_c} p(co_j) \tag{23}$$

In Fig. 3 case, we had $p(C_1) \geq p(C_2)$, which resulted in agglomeration of C_2 components under the cluster C_1.

4.2.3 Statistical neighbourhood processing

Every time component grows, or outlier gets promoted to a component, we need to test and process component statistical neighbourhood. Since the statistical distance does not have the same properties as Euclidean, using some well-established mechanisms such as M-trees (Ciaccia et al. 1997) cannot be used. The statistical distance is always relative, observed from a specific component or population. This would mean that calculating the component statistical neighbourhood in a naïve attempt would require to calculate the statistical distance to all other classification objects, which would be very expensive. As part of the classification we already detect statistical neighbourhood in (19). Using Q_n on $co_c \in Cm$ we can define statistical neighbourhood as

$$N(co_c) \leftarrow N(co_c) \cup Q_n \tag{24}$$

Outlier inclusion: Each time the classified component co_c grows, a nearby outlier can join its population if the outlier statistical distance to the center of the component is $\leq \theta$. We define the outlier inclusion as

$$\forall o_i \in (N(co_c) \cap O) : d_\sigma(\mu(o_i), \mu(co_c), \Sigma(co_c)) \leq \theta \tag{25}$$

This means iterating through the classified component co_c statistical neighbourhood $N(co_c)$ to re-check outlier statistical distances. If any of the outliers satisfies the θ-bound component membership defined in (9) we add it to the component co_c and remove it from the container tree G_c and outlier set O.

Component inclusion: Sometimes components can overlap in such extent that center of one component falls in the population of the other component. In this case, both components share the same population. We can discover such situation when iterating through the statistical neighbourhood of the classified component co_c. Each component in the statistical neighbourhood of the classified component $N(co_c)$, whose center is within θ statistical distance of the component co_c center, are called *included components*. Such components are redirected to the classified component co_c by updating their redirection target component

$$\forall co_i \in (N(co_c) \cap Cm) : d_\sigma(\mu(co_i), \mu(co_c), \Sigma(co_c)) \leq \theta \Rightarrow R(co_i) \leftarrow co_c \tag{26}$$

When updating population of the classified component co_c that has been redirected, i.e., $R(co_c) \neq \emptyset$, the redirection target component $R(co_c)$ is updated instead. The redirection exists as long as both of the involved components exists. By removing one of the involved components, we remove the redirection between them as well. Also, both components involved in the redirection get agglomerated under the same cluster no matter how much shared population members they have.

Such mechanism obviously stops all updates on the redirected component $co_c \in Cm$ and enables the population of the redirection target component $R(co_c)$ to grow. Once the population of the redirected component falls significantly behind the population of the redirection target component

$$p(co_c) < p_{rr} p(R(co_c)) \tag{27}$$

we can declare the redirected component co_c as obsolete, and remove it as soon as possible. We introduce an SHC parameter p_{rr} which represents the allowed ratio between populations of the redirected and redirection target components.

4.2.4 Statistical agglomeration example

In Fig. 4 we can see an example of statistical agglomeration over a single population of 200 data objects, defined with variance ${\sigma_{(i,i)}}^2 = 13$. The statistical agglomeration processing was done in 7 slices, seen in Figs. 4a–g. Figure 4h shows the final result with all population members.

After the first slice, in Fig. 4a, we can see a number of θ-bound components and outliers. The virtual covariance (10) gives outliers circular statistical area around them. In Fig. 4b we can already observe outlier and component inclusions, and shared population member agglomerations. Components at this point do not have big populations, therefore it is easy to include them in bigger populations. Component inclusions happening from Fig. 4c take more time, since at this point we have bigger components. All component inclusions are done in Fig. 4g, where we look at the final component, modelling the example population. In Fig. 4g we can see occurrence of an outlier, which would be probably included in the final component as the input data stream continues. This is a static example, since the population does not evolve over time.

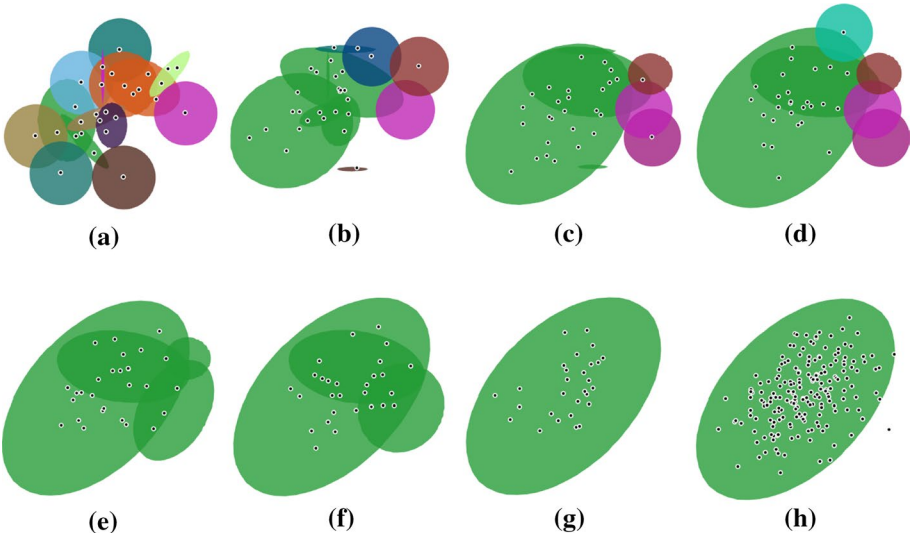


Fig. 4 Statistical agglomeration example

4.3 Population drift and split

Population evolution can lead to uncontrolled component growth. Such growth can be limited by (12), or by adding population drift and split mechanism we can closely track the population evolution. After promoting an outlier into a component, the newly formed component needs to acquire enough members to become stable. In the moment when a component becomes stable we can take its snapshot, named *component baseline*. A component baseline can be taken when a component gets specific number of population members, or grows above certain variance (12).

We define SHC parameters p_{cb} as the minimal component population and ${\sigma_{cb}}^2$ and the minimal component variance needed for a component to become stable. If $p_{cb} = 0$ and ${\sigma_{cb}}^2 = 0$, SHC component drift and split tracking is switched off.

The component baseline is taken after the component co_c is updated with newly retrieved data object X^{m+1} from the input data stream

$$p(co_c) \geq p_{cb} \vee \theta_{lim}(\Sigma(co_c), {\sigma_{cb}}^2) = 1 \Rightarrow cb(co_c) \leftarrow co_c \tag{28}$$

The novel concept introduced in SHC is using a hierarchy of clustering algorithm instances for tracking and analysing component drift and split. Each component has a child SHC instance that can be used for periodic analysis whether the component population evolves in such way it starts drifting from the original position, or splitting in several distinct sub-populations. In Fig. 5a we can see a timeline of the component co_c. After the component initialization at time t_1, we need to wait until the component baseline $cb(co_c)$ is taken. Then we can use the child SHC instance $SHC(co_c)$ to sub-cluster a window of newly added data objects to the component. In Fig. 5a this sub-clustering is occurring between time points t_2 and t. We can take that t is the current time and the moment when X^{m+1} was retrieved from the input data stream. In this window, the child SHC instance can capture a

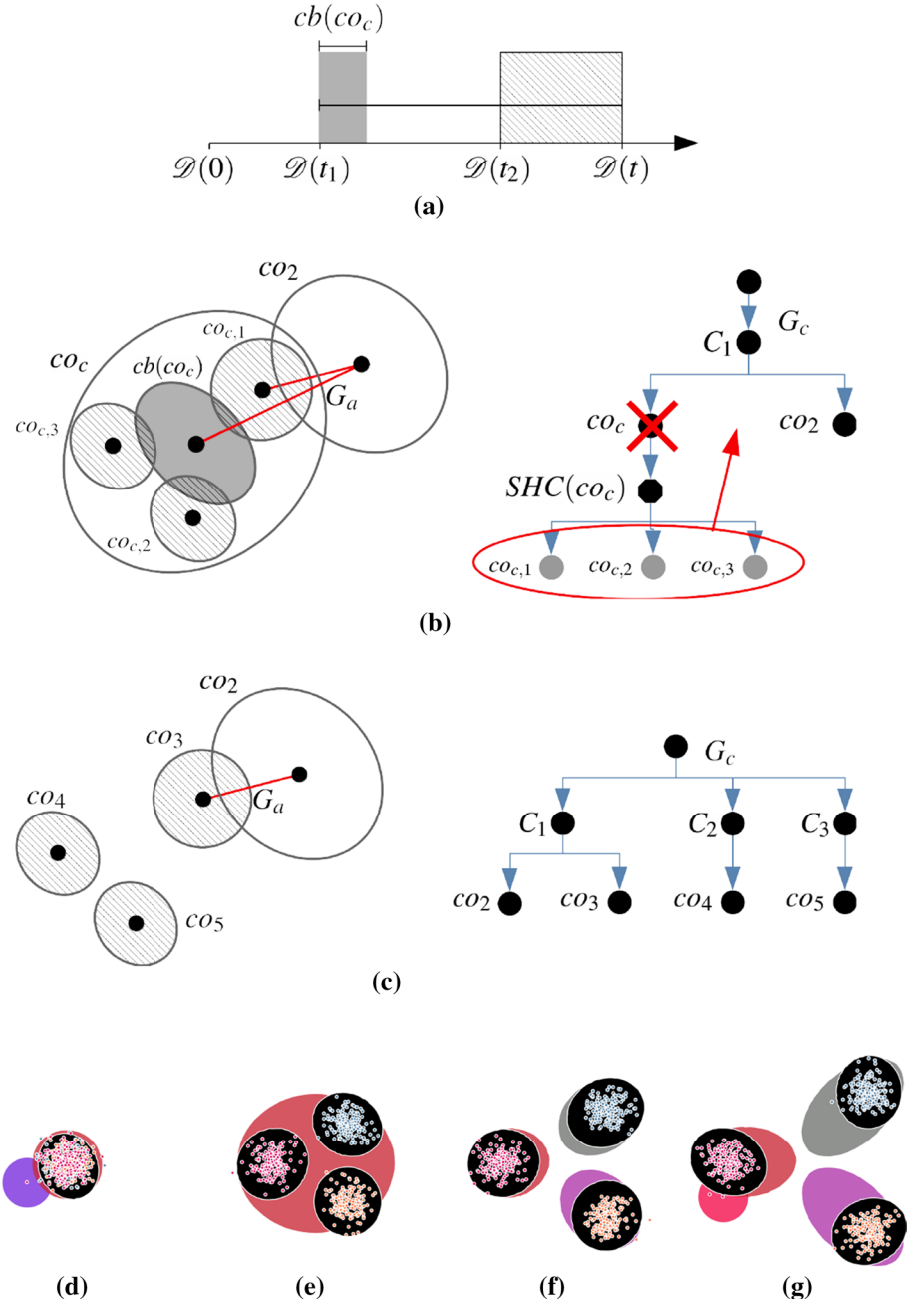


Fig. 5 Sub-clustering in component drift and split

number of components, depending on how the component co_c population evolves. Assuming that data object X^{m+1} at time t updated the component co_c, we can start analysing

component population drift and split by selecting the biggest **component** of the child SHC instance $SHC(co_c)$.

$$co_{c,biggest} = \underset{co_{c,i} \in Cm(SHC(co_c))}{arg\,max}\ p(co_{c,i}) \tag{29}$$

Outliers in the child SHC instance are not taken in account as they do not represent the component evolution. We additionally define a *drift index* as

$$d_i(co_c) = \sum_{co_{c,i} \in Cm(SHC(co_c))} p(co_{c,i}) d_\sigma(\mu(co_{c,i}), \mu(cb(co_c)), \Sigma(cb(co_c))) \tag{30}$$

which represents weighted average statistical distance between component baseline center and sub-clustered component centers. Weights used in the drift index calculation are sub-clustered components populations, so that bigger components have bigger contribution to the drift index.

For drifting we define two SHC parameters: *drifting size ratio* p_d and *drifting index ratio* θ_d. The drifting size ratio is used to check the population ratio between the biggest sub-clustered component and the component co_c baseline. Tracking and analysing component drift and split by sub-clustering newly added data objects to the component co_c is done until the size of the biggest sub-clustered component reaches

$$p(co_{c,biggest}) \geq p_d p(cb(co_c)) \tag{31}$$

At this moment the decision about component drift and split is taken by evaluating drift index d_i. The drifting index ratio is used to check the ratio between drifting index and statistical distance threshold θ.

Component drifting or splitting detected: After the size condition in (31) is met, we consider component drifting or splitting detected when

$$di(co_c) \geq \theta_d \theta \tag{32}$$

In Fig. 5b we can see an example of a component drift and split into three distinct sub-populations. The agglomeration graph G_a between ordinary and sub-clustered components must be kept updated at all times. This is possible due to having Q_c (19) for classifying new data object X^{m+1} in both parent and child SHC instances. The newly processed data object X^{m+1} is a shared population member between normal and sub-clustered components for

$$E_{a_{sub}} = ((Q_c \cap Cm) \setminus \{co_c\}) \times (Q_c(SHC(co_c)) \cap Cm(SHC(co_c))) \tag{33}$$

Pairs of normal and sub-clustered components from $E_{a_{sub}}$ create agglomeration graph edges as defined in (21). After detecting component drift or split, we adjust the container tree G_c by moving sub-clustered components from the child SHC instance into normal components and declaring the component co_c obsolete. In such move, we retain the edges of the agglomeration graph G_a between normal and sub-clustered components that are according to (21). Figure 5c shows the final result, after moving the sub-clustered components into normal components, removing the component co_c and restructuring the container tree G_c for all component connections in the agglomeration graph G_a that are above the threshold θ_{sp} (22). Figure 5d–g show component drift and split example processing done by SHC. Sub-clustered components are marked with black background.

Component drifting or splitting NOT detected After the size condition in (31) is met, we consider that component is NOT drifting or splitting when

$$di(cm_c) < \theta_d \theta \tag{34}$$

In this case, we can remove the child SHC instance $SHC(co_c)$ and all of the sub-clustered components. Component drift and split tracking can be delayed for $p(cb(co_c))$ new data objects, after which we can start performing sub-clustering again.

4.4 Decay mechanism

We define an SHC decay parameter δ. If a classifier object $co \in Co$ did not update after

$$\forall cm \in Cm : \delta \min(\max(p_{cb}, p(cm)), \delta_{max} p_{cb}), \forall o \in O : \delta_o \delta p_{cb} \tag{35}$$

data objects retrieved from the input data stream, it is marked as obsolete and eventually removed. Component decay depends on the population size within the interval $[p_{cb}, \delta_{max} p_{cb}]$. Outliers decay at $\delta_o = 20$ times lower pace, as we give them a chance to get promoted to components, or get examined by user in search for fraudulent behaviour. If $\delta = 0$, SHC does not use the decay mechanism, i.e., classification objects do not decay.

4.5 Removing obsolete classification objects

Obsolete classification objects get removed eventually, which can result in container tree G_c and agglomeration graph G_a restructuring. The agglomeration graph G_a is disconnected, having as many connected subgraphs as the number of clusters. Having K clusters in G_c means that G_a can be separated into $M \leq K$ connected subgraphs, taking in consideration threshold θ_{sp}. Components N_a (16) can be separated into a set of partitions

$$N_a = \{N_{a_1}, N_{a_2}, ..., N_{a_M}\} \tag{36}$$

where each partition is a set of connected components under the same cluster. The obsolete component co_o is a member of partition N_{a_i}

$$co_o \in N_{a_i} \tag{37}$$

By removing co_o, we test the partition N_{a_i}, and if it can be further partitioned into a set of new L partitions

$$N_{a_i} = \{N_{a_{i,1}}, N_{a_{i,2}}, ..., N_{a_{i,L}}\} \tag{38}$$

we can conclude that due to removal of the obsolete component co_o, cluster C_i divided into L separate clusters, which must be reflected in the container tree G_c. Figure 6 shows such a situation. In Fig. 6a we can see seven connected components, all under the same cluster C_1. By removing component co_3, our partition $N_{a_1} = \{co_1, co_2, ..., co_7\}$ is fragmented into $L = 3$ new partitions $N_{a_{1,1}} = \{co_1, co_2\}, N_{a_{1,2}} = \{co_4, co_5\}, N_{a_{1,3}} = \{co_6, co_7\}$. This results in division of the cluster C_1 into clusters $\{C_1, C_2, C_3\}$ as seen in Fig. 6b.

Removing an obsolete outlier does not affect the agglomeration graph G_a and requires only removals in the container tree G_c.

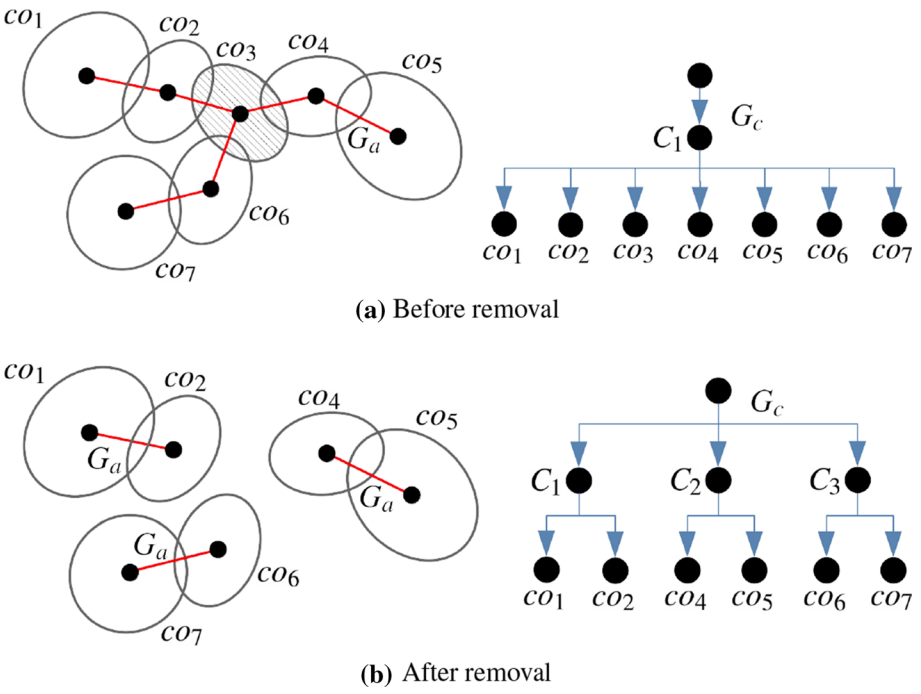


Fig. 6 Removing obsolete components, traceability tree

4.6 Traceability

To support traceability, we need a *traceability tree* G_t for all obsolete classification objects. The traceability tree is an acyclic directed graph whose nodes are obsolete and current classification objects. Edges of the *traceability tree* form a transitional time line, which helps us reconstruct transitions between obsolete classification objects Ob and current classification objects $Cl \cup Co$. Current classification objects $Cl \cup Co$ are leafs of the traceability tree G_t. Once a current classification object becomes obsolete, a transition between it and the new current classification object is created.

$$\begin{aligned} &G_t = (N_t, E_t), Ob = N_t \setminus (Cl \cup Co) \\ &\forall (nt_i, nt_j) \in E_t \nexists (nt_j, nt_k) \in E_t : nt_i \in Ob \wedge nt_j \in (Cl \cup Co) \end{aligned} \tag{39}$$

A classification object $nt_i \in Ob$ is a predecessor of the current classification object $nt_j \in (Cl \cup Co)$ if there is a directed path in G_t from node nt_i to node nt_j, or $P_t(G_t) = nt_i - nt_j$. The drift and split example in Fig. 5 creates the traceability tree G_t seen in Fig. 7.

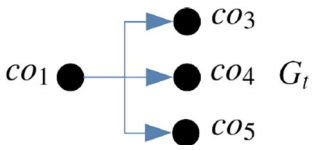


Fig. 7 A traceability tree

4.7 SHC algorithm details

4.7.1 SHC input parameters

Table 1 contains all SHC parameters. Parameters are divided into several sections: general, limitation, and drift, depending on where and how parameters are used. General SHC parameters are used always. There are some significant notable parameter values:

- $\delta = 0$: no decay
- ${\sigma_{\theta_{min}}}^2 = 0$: no variance limit when promoting outliers to components
- ${\sigma_{\theta_{max}}}^2 = 0$: no variance limit on component growth
- $p_{cb} = 0 \wedge {\sigma_{cb}}^2 = 0$: analysing drift and split is switched off

SHC limitation and drifting mechanisms are generally not needed unless we have some specific dataset or data stream.

To make SHC more configurable and less parameter-dependent, we introduced several generic parameter templates. Table 2 gives some agglomeration related templates. *Aggressive* agglomeration template tends to agglomerate more population members than the *Relaxed* agglomeration template. Similarly, Table 3 contains drifting templates.

4.7.2 SHC detailed description

SHC is a single-pass algorithm, which means that each time a new data object X is retrieved from the input data stream, SHC undertakes the same set of steps to return the classification. Figure 8 shows a high-level overview of SHC. On the left side in Fig. 8, we can see the major steps of the algorithm, and each step detailed on the right side. We additionally give detailed pseudo-code for each of the major steps in the Appendices. Throughout the processing SHC maintains the container tree G_c (13), agglomeration graph G_a (16), and traceability tree G_t (39). All classification objects (17,18) are kept in the container tree G_c. The main procedure pseudo-code is given in Appendix A. Here we give brief descriptions of each step in the main procedure.

Step 1: Decay This step is described in Sect. 4.4, and pseudo-code is given in Appendix B. We iterate through all classification objects $\forall co \in Co$ and decrement their decay counters $\delta(co)$ (17, 18). After decrementing, all classification objects that have expired decay counter $\delta(co) \leq 0$ are marked obsolete.

Step 2: Classification This step is described in Sect. 4.2.1, and pseudo-code is given in Appendix C. In this step we perform a query on classification objects Co that are not obsolete and are below the variance limit threshold ${\sigma_{\theta_{max}}}^2$ (12). In a naïve query, we iterate through all these classification objects and create the classified set Q_c and neighbourhood set Q_n as described in (19). Once the classification is done, we search for

Table 1 SHC main parameters

Parameter	Name	Default value	Details
General			
θ	Statistical distance bound	3.0	**Mandatory.** Statistical distance that makes component or outlier classification bound (9).
${\sigma_v}^2$	Virtual variance	0.5	**Mandatory.** Variance used to create outlier virtual covariance matrix (10).
θ_{sp}	Shared population threshold	1	**Mandatory.** Defines how many shared population member is needed for connected component agglomeration (21).
p_{rr}	Redirected component removal ratio	0.2	**Mandatory.** Ratio between redirected and redirection target components that initiate redirected component removal as defined in Sect. 4.2.3.
p_n	Statistical neighbourhood distance	3	*Optional.* Maximal statistical neighbourhood distance as defined in (19).
δ	Decay	10	*Optional.* Decay parameter as defined in Sect. 4.4.
Limitation			
${\sigma_{\theta_{min}}}^2$	Minimal variance for components	$0.1 * {\sigma_v}^2$	*Optional.* Minimal variance needed for promoting outliers to component (12).
${\sigma_{\theta_{max}}}^2$	Maximal variance for components	0	*Optional.* Maximal allowed variance for components (12).
Drift			
p_{cb}	Component baseline minimal population	40	*Optional.* Minimal population for defining the component baseline (28).
${\sigma_{cb}}^2$	Component baseline minimal variance	8	*Optional.* Minimal variance for defining the component baseline (28).
p_d	Drifting size ratio	0.7	*Optional.* Used to detect whether drifting assessment can be performed, as defined in Sect. 4.3.
θ_d	Drifting index ratio	0.5	*Optional.* Used to detect drifting. Defined in Sect. 4.3.

Table 2 SHC agglomeration templates

Parameter	Aggressive	Normal	Relaxed
θ	3.5	3.2	2.9
${\sigma_v}^2$	1.2	1.0	0.8
p_n	4	2	2

Table 3 SHC drifting templates

Parameter	No drift	Slow drift	Normal drift	Fast drift	Ultra fast drift
p_{cb}	0	120	80	40	20
${\sigma_{cb}}^2$	0	9	8	6	6
p_d	-	2.0	1.3	1.0	0.5
θ_d	-	1.0	0.8	0.6	0.5

the classified object in Q_c whose center is statistically the closest to the new data object (20).

Step 3: Agglomeration We perform the shared population agglomeration described in Sect. 4.2.2. Detailed pseudo-code is given in Appendix D. The shared population agglomeration is done on the previously created classified set Q_c. The agglomeration function receives pairs of classified objects from Q_c, as defined in (22). If one of the classified objects in the agglomerated pair is an outlier, we include this outlier in the other classified object by updating its model according to (8). If both classified objects are components, we need to create or update a connection edge between them in the agglomeration graph G_a, and increment the shared population counter sp, as defined in (22). Only when shared population member counter sp reaches the threshold θ_{sp}, we restructure the container tree G_c so that both components are under the same cluster. In this case, we agglomerate under the bigger cluster (23). If the original closest classified object co_c was an outlier, it is certain that it was removed by the agglomeration procedure, hence, we need to find a new closest classification object co_c, which is updated in the next step. A smart implementation keeps the statistical distances calculated in the classification step as part of the classified set Q_c. This way, we need to recalculate the statistical distance to the new data object X only for components that were updated in the agglomeration procedure and save significant processing time.

Step 4: Model update The model update is described in Sect. 3. Detailed pseudo-code is given in Appendix E. If co_c is an outlier, we promote it to a component only if the newly formed component would have the variance (12) $> {\sigma_{\theta_{min}}}^2$. If co_c is a component, we update it only if its variance is $< {\sigma_{\theta_{max}}}^2$. Besides the model update according to (8), we need to increment the population counter $p(co_c)$ (17,18) and reset the decay counter $\delta(co_c)$.

Drift and split If the component baseline is not formed, we track whether the population counter $p(co_c)$ and component variance $\Sigma(co_c)$ satisfy the conditions defined in (28). When these conditions are satisfied, we clone the current component model into the baseline $cb(co_c) \leftarrow co_c$. Drift and split evaluation is done periodically, done by implementing a component population counter. A child SHC is instanced every time we begin drift and split evaluation and data objects that updated the co_c component model are passed to the child SHC instance. We keep connections between normal and

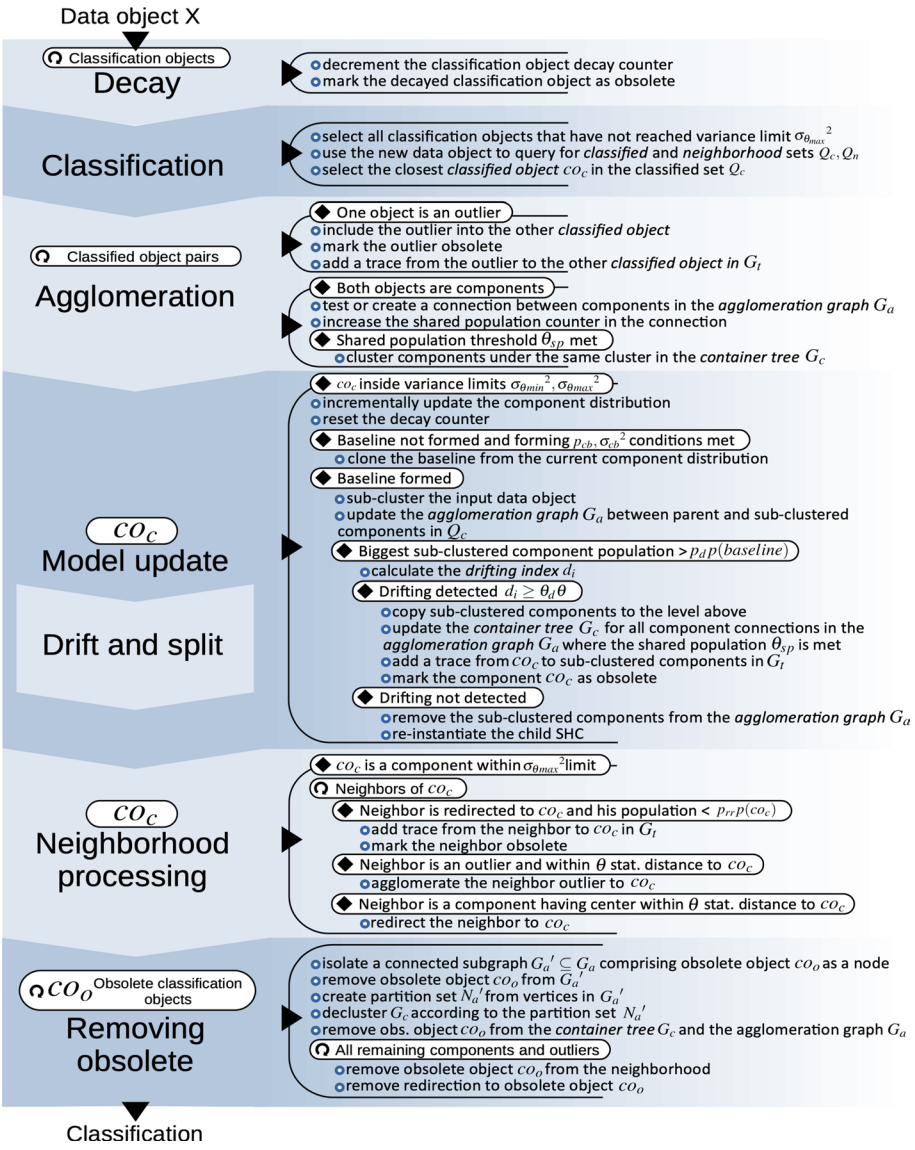


Fig. 8 SHC overview

sub-clustered components at any time when we perform drift and split evaluation. We do it as defined in (33). Once the biggest sub-clustered component (29) reaches the threshold (31), we are certain that the population evolution can be correctly assessed. Drift and split is then assessed by calculating drift index *di* (30). If drift and split was not detected (34), we remove the child SHC, clean the agglomeration graph G_a, and wait another number of component data objects to begin another drift and split evaluation. If drift and split was detected (32), we move sub-clustered components into normal components and simultaneously cluster in cases where there is a sufficient number of shared

population members (22). After we moved all the components, we make the updated component co_c obsolete and remove the child SHC.

Step 5: Neighbourhood processing We process the statistical neighbourhood of the closest classified object co_c. The neighbourhood processing is described in Sect. 4.2.3 and pseudo-code is given in Appendix F. Only components can have neighbourhood (19). We process neighbourhood for components that have variance $< {\sigma_{\theta_{max}}}^2$. For all component neighbours, we check whether the neighbour center is within the component θ bound. If the neighbour is an outlier, we perform the outlier inclusion (25) by adding it to the component model. If the neighbour is another component, we perform the component inclusion (26) by redirecting the neighbour model updates to the component co_c. Eventually, if the redirected neighbour falls behind the component co_c in the population size (27), we mark the redirected neighbour as obsolete.

Step 6: Removing obsolete The whole removal process is described in Sect. 4.5. Detailed pseudo-code is given in Appendix G. We remove all classification objects that were marked as obsolete co_o. First we try to find a vertex-induced subgraph ${G_a}'$ of the agglomeration graph G_a that comprises the obsolete classification object co_o. If we are able to find such a subgraph, we remove the obsolete classification object co_o from it. After this, we try to partition nodes of the subgraph, and if the partitioning results in several new disjunct partitions (38), we need to decluster all the partitions in the container tree G_c. The rest of the removal procedure is performing cleaning after removing co_o, such as the agglomeration graph G_a, container tree G_c, other component neighbourhoods and redirections.

4.8 Computational complexity

SHC is made of several steps, which all have their own computational complexities. To asses the SHC complexity means summing complexities of all individual steps.

The decay mechanism iterates over all classification objects. The number of classification objects $k = |Co|$ is limited by SHC parameters. The calculation of the worst case scenario for k is given in Appendix H and can be taken as $k = \delta_o \delta p_{cb}$ for the maximal number of outliers, which greatly affect later neighbourhood processing. For the decay mechanism we have complexity $O(k)$. However, a decay counter recalculation is a simple and fast calculation, which we did not want to make more complex by introducing additional auxiliary structures.

For the naïve query approach we used, the complexity is $O(k)$. By using some of the indexing techniques, this can be reduced to $O(\log k)$. The query could be also parallelized, which would bring its complexity down to $O(1)$ given that we have at least k parallel nodes. The most important results of the query are classified and neighbourhood sets, Q_c, Q_n, whose results directly affect the subsequent SHC phases. The maximal number of potential classified objects in the Q_c can be resolved by the kissing number k_2, which is detailed in the Appendix I (Conway and Sloane 2013). The most classified objects we can get in the classified set Q_c is k_2 outliers, as they can be packed densely. Any combination of components and outliers will certainly be less than k_2, which sets the complexity of the agglomeration procedure to $O(k_2)$.

When calculating the maximal neighbourhood Q_n, we estimate that the worst case scenario is to have only outliers in the neighbourhood. This leads to conclusion that assessing the potential neighbourhood is a combination of the sphere-packing and neighbourhood size problems. Simple method includes assessing the maximal number of outliers in the neighbourhood that would not be included in the classified component according to the

normal distribution. For $p_n = 3$, we can take that a component could assimilate its neighbourhood when the number of outliers in the neighbourhood is greater than $\Phi_{\mu=0,\sigma=1}(-1)$, where Φ is the cumulative normal distribution function, which could be the good approximation of the maximal outlier number that could still be taken as the neighbourhood, and not as part of the classified component co_c. Thus, the maximal neighbourhood can be calculated as $k_3 = \Phi_{\mu=0,\sigma=1}(-1) \underset{co_c \in Cm}{max} p(co_c)$, and the complexity of the neighbourhood processing procedure is $O(k_3)$.

The updating procedure is done for the classified component co_c, hence, the basic complexity is $O(1)$. In case of the drift and split evaluation, the worst case in the child SHC is to have only outliers. In particular this is $k_4 = p_n \underset{co_c \in Cm}{max} p(co_c)$ outliers, which sets the updating procedure complexity to $O(k_4)$.

Finally, the removal procedure must remove all obsolete classification objects. This can be due to decay of a classified object or splitting of a component. According to the decay function (35), in each step we can have at most one outlier and at most $(\delta_{max} - 1)p_{cb}$ components that are obsolete due to decay. If we add one potential splitting classified component, the complexity of the removal procedure is $O(2 + (\delta_{max} - 1)p_{cb})$.

The overall complexity of SHC estimates at

$$\begin{aligned} &O(m(2k+2)) \leq O(m(2k + k_2 + k_3 + k_4 + (\delta_{max} - 1)p_{cb} + 2)) < O(m(6k+2)) \\ &6k + 2 \ll m \end{aligned} \tag{40}$$

which is positioning SHC well below the *k-means* complexity. However, if we check one of the similar algorithms, such as DBSTREAM (Hahsler and Bolaños 2016), SHC has slightly higher complexity. Adjusting some procedures, and by introducing additional auxiliary structures, we could lower the SHC computational complexity even more.

5 Evaluation

The proposed SHC algorithm is implemented in the C++ programming language. We used Eigen (Jacob et al. 2013) to support algebraic calculations for the multivariate normal distribution. The C++ code was interfaced to R using *Rcpp* R package (Eddelbuettel and Balamuta 2018).

For the evaluation purposes, we have used the *stream* and *streamMOA* R packages (Hahsler et al. 2017, 2018). These packages provide various data stream clustering algorithms such as BIRCH, BICO, DenStream, CluStream, ClusTree, D-Stream, DBSTREAM, and others. Additionally, these two packages provide many clustering indices (Desgraupes 2017) that can be used to compare clustering algorithms, as already done by the previous work (Carnein et al. 2017). *streamMOA* R package is only an interface between MOA package (Bifet et al. 2018) written in Java and *stream* R package, providing access to MOA clustering algorithms. We extended the *streamMOA* R package to provide access to the MCOD clustering algorithm available in the MOA Java package.

We have created an interface between SHC R interface and abstract classes in the *stream* package, to allow testing all relevant clustering algorithms on the same test cases and data streams. Since the *stream* package was not intended for the outlier testing, given synthetic data stream generators did not generate and label outliers and there were no outlier accuracy indices implemented. We extended the *stream* package to add these features.

We used CRI for most of the algorithm comparisons, which shows to be a very good external clustering indicator. In some cases, we used the purity clustering index. However, both CRI and purity do not consider or penalize outlier mismatches, which are important for this paper. Even if a clustering algorithm does not recognize any of the outliers from the input stream, the difference in the final CRI and *purity* values is minor because of the small outlier proportion and does not reflect the algorithm difference in the outlier recognition accuracy. Since the extended synthetic data stream generator is capable of labeling generated outliers, we applied *Jaccard Index* (Jaccard 1912; Desgraupes 2017) for the outlier detection accuracy assessment. We used *true positive* (TP), *false positive* (FP) and *undetected* (UND) for elementary detections when calculating the *Outlier Jaccard Index* (OJI)

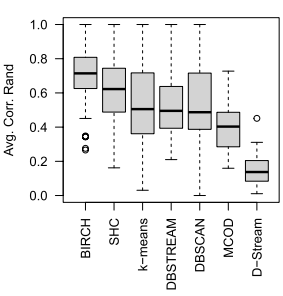
$$OJI(\text{generated}, \text{detected}) = \frac{|\text{generated} \cap \text{detected}|}{|\text{generated} \cup \text{detected}|} = \frac{TP}{TP + FP + UND} \tag{41}$$

which we also implemented in R and added to the indices in the *stream* R package. Assessing whether a tested clustering algorithm correctly recognizes a marked outlier turns out to be quite ambiguous since many clustering algorithms do not mark outliers explicitly. In fact, the only algorithm that explicitly marks detected outliers, except SHC, is MCOD. For others, we considered a macro-cluster, or a micro-cluster for *BIRCH*, comprising only the marked outlier, as a minimal condition and evidence that the tested clustering algorithm can isolate outliers and anomalies even if it is not explicitly designed for the outlier recognition.

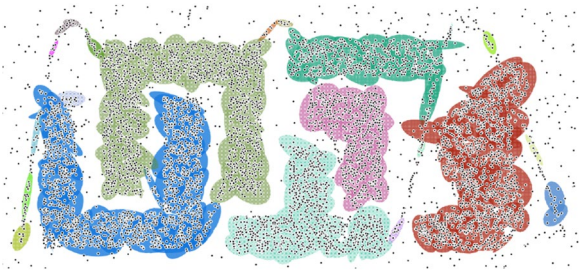
For data streams, all tests were done in small data object batches. First, we performed only classification for a batch of data objects, then we did the model update for the same batch. The size of the data stream batch was variable and depended on the tested clustering algorithm. Of all tested clustering algorithms, SHC is the only single-phase algorithm, providing us with the ability to classify and update model for each data object separately. In this case, the coincidence matrix needed for calculating CRI, purity, and OJI was updated gradually, after each data object classification. For all other clustering algorithms, a bigger data stream batches were selected, as suggested by the *stream* package authors (Hahsler et al. 2017), which made data stream processing more coarse than for the SHC algorithm. This is due to the fact that the final classification and macro-cluster assignments are obtained only after invoking the offline phase, which greatly affects how the clustering indices are calculated, i.e., they need to be calculated for the whole data stream batch at once.

Calculating OJI for continuous data streams turns out to be an incremental problem. After processing each data stream subset, we obtain TP, FP, and UND values for that subset only. We need to sum these values continuously for all previously processed data stream subsets. OJI can be calculated from these values at any point in the processed data stream, which makes it incrementally cumulative when processing data streams. In the testing, we call it *cumulative Outlier Jaccard Index* (cOJI). cOJI can show outlier detection accuracy trend throughout data stream processing.

For the hyper-parameter tuning, we used model-based Bayesian optimization (MBO). For this purpose we used *mlrMBO* R package (Bischl et al. 2017; Horn et al. 2015). Hyper-parameter tuning was done for each test case separately.



(a) SHC seeds dataset testing results



(b) Chameleon DS3 SHC clustering

Fig. 9 Seeds and Chameleon DS3 datasets clustering

Table 4 Algorithm parameters for the seeds dataset

Algorithm	Setting
SHC	$\theta = 7, \sigma_v{}^2 = 2.87, \theta_{sp} = 24, \delta = 0, \sigma_{\theta_{min}}{}^2 = 0.58, \sigma_{\theta_{max}}{}^2 = 0.604$
DBSTREAM	$r = 3.11, Cm = 0.657, shared\_density = true$
DBCAN	$eps = 1.33$
BIRCH	$T = 3, branching = 3, maxLeaf = 3$
D-Stream	$gridsize = 1.15, Cm = 1.19$
MCOD	$r = 4.17, t = 15, k = 3$

5.1 Test case 1: Static testing with Seeds dataset

For this test, we obtained the Seeds dataset (Dua and Graff 2017). This small dataset comprises 210 measurements of 3 distinct seed types. Each seed type is measured 70 times. It is a small dataset that has 3 significantly overlapping clusters. This dataset favors batch clustering algorithms and data stream clustering algorithms that can handle cluster overlapping, such as DBSTREAM and SHC. The dataset is sorted by the seeds type and must be manipulated for usage in clustering algorithm testing. Therefore, we performed 7-fold cross-validation testing on this dataset. 10 seeds of each type were used as a learning set for all clustering algorithms. The rest of the 60 seeds of each type were used as a testing set. We repeated this for all 7 learning sets and calculated average CRI for all tests. Each algorithm went through the hyper-parameter tuning to obtain the best results. Parameters obtained this way are available in Table 4.

In Fig. 9a we can see the testing results for the seeds dataset. In this testing, BIRCH seems to be the best clustering algorithm, followed by SHC and k-means. All other clustering algorithms that are not in Fig. 9a did not show any significant results for this test case.

5.2 Test case 2: Chameleon DS3 dataset

For testing SHC variance limitation capability, we selected the Chameleon DS3 dataset (Karypis et al. 1999). SHC was initialized manually by using $\theta = 2.84, \sigma_v{}^2 = 0.4, \sigma_{\theta_{min}}{}^2 = 0.1, \sigma_{\theta_{max}}{}^2 = 6.5$. The final result was cleaned by removing

Table 5 Algorithm parameters for the synthetic clusters and outliers

Algorithm	Setting
SHC agg.	SHC aggressive template in Table 2
SHC nor.	SHC normal template in Table 2
SHC rel.	SHC relaxed template in Table 2
DBSTREAM	$r = 2.833, Cm = 0.766, lambda = 0$
DBSCAN	$eps = 1.47$
BIRCH out.	$T = 1.833, branching = 520, maxLeaf = 520$
BIRCH clus.	$T = 4.927, branching = 520, maxLeaf = 520$
D-Stream	$gridsize = 1.017, Cm = 0.1, Cl = 0.02$
MCOD	$r = 7.3, t = 100$

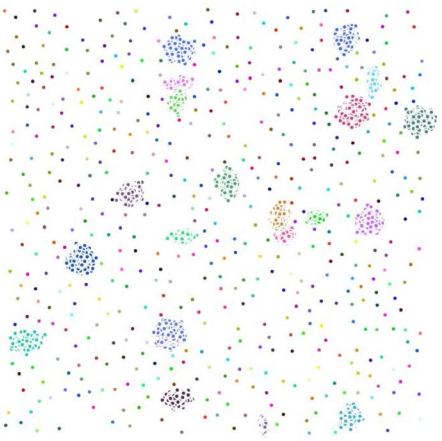
outliers and components having small populations $\forall cm \in Cm : p(cm) < 7$. The final clustering result can be seen in Fig. 9b.

5.3 Test case 3: Synthetically generated statistic clusters and outliers

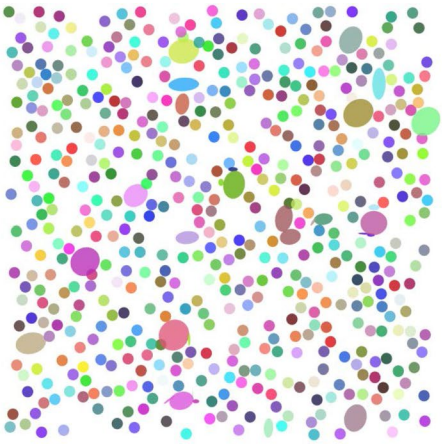
We used a generic synthetic stream generator, capable of generating normally distributed populations. In this test case, we generated 20 normally distributed clusters and 500 outliers, forming a data stream made of 30000 data objects. All cluster populations had variance $\sigma^2 \in [0.5, 8]$, and outliers were generated to be within ${\sigma_v}^2 = 1$. Covariance correlation was also randomly selected for each cluster in values $\rho_{(i,j)} \in [0, 0.2]$, as we wanted our components to be more spherical. The whole space was limited to $(0, 0) - (240, 240)$. For smaller space than that, we would not be able to generate 20 clusters and 500 outliers statistically correctly distributed, so that the generated problem is solvable by the clustering algorithms. We generated 10 random datasets using previously defined parameters and processed them by the compared clustering algorithms. This means 300000 data objects in total for all 10 generated datasets. The final displayed result is the average of all 10 random tests.

In Table 5 we can see parameters tuned for the synthetically generated dataset. For SHC, we used templates from Table 2, to compare how different parameters affect the final clustering result. All clustering algorithms, except MCOD and BIRCH, were multi-objective hyper-parameter tuned for CRI and OJI, as we wanted to get the best possible result in both clustering and outlier detection. We were unable to do such tuning for MCOD since micro-clusters are tied to the radius r, which made MCOD performing poorly when multi-objective tuned. Therefore, MCOD was tuned only for OJI, as it is a specialized outlier detection algorithm. We had a similar issue with BIRCH, however, we created two distinct settings for BIRCH, one tuned for cluster detection, and one for outlier detection.

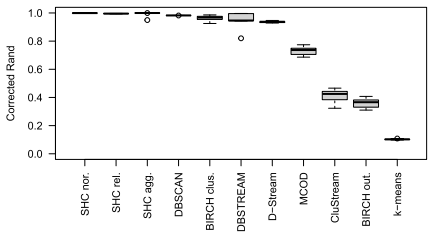
In Fig. 10a we can see an example of the generated dataset. We can visually identify clusters among points that look like noise. In Fig. 10b we can see the model generated by SHC parametrized with *normal agglomeration template* from Table 2. Average CRI is available in Fig. 10c. In this case SHC is the most accurate algorithm. Since we tuned MCOD for best outlier detection results, its CRI is not among the best algorithms. Figure 10d presents average OJI results for all algorithms. As expected, MCOD achieves excellent results in the outlier detection context. It is interesting to observe the difference between the BIRCH cluster and outlier setting. Whereas BIRCH cluster setting achieves very high CRI, the outlier setting achieves higher OJI. We would get similar,



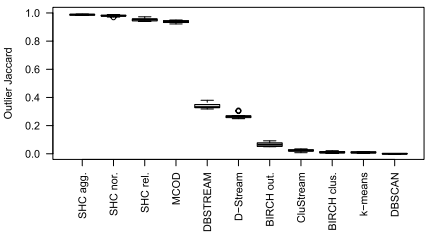
(a) Synthetically generated dataset example



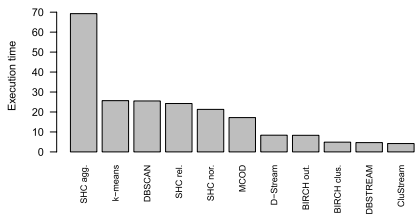
(b) SHC model for the example



(c) Corrected Rand results



(d) Outlier Jaccard Index



(e) Execution times (s)

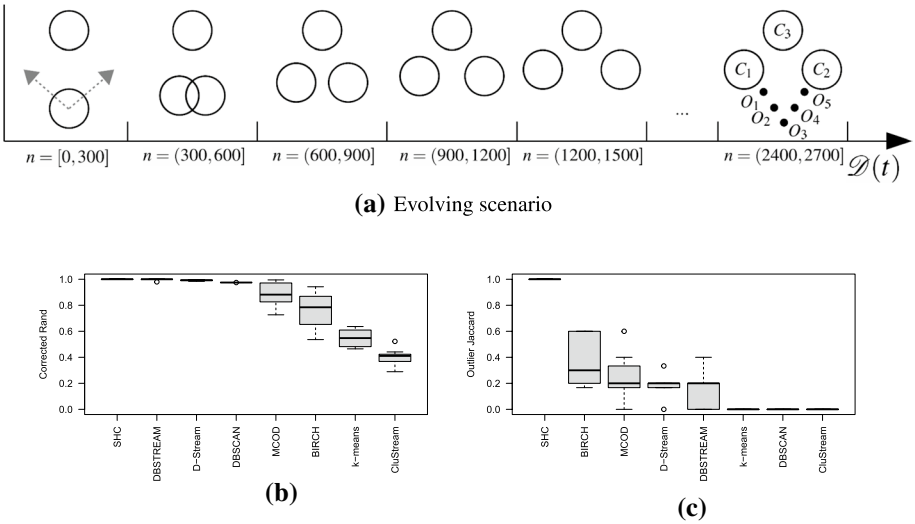
Fig. 10 Synthetically generated clusters and outliers results

but poorer results for MCOD, since MCOD operates only on the micro-cluster level and has no macro-cluster related second stage. We did not add a second stage for MCOD as it would only diminish good outlier detection results.

As we increase proportion of outliers in the overall dataset, we can see that standard clustering algorithms, such as *k-means*, start performing very poorly. This is a clear sign that more complex clustering algorithms are needed to deal with modern data science challenges. Modern hardware is quite adequate for such a raise of the clustering algorithms complexity, which can be seen in Fig. 10e. All tests were conducted on 8 core

Table 6 Algorithm parameters for the evolving data stream test

Algorithm	Setting
SHC	SHC normal agglo. template in Table 2 and fast drift template in Table 3
DBSTREAM	$r = 4, Cm = 0.209, shared\_density = true$
DBSCAN	$eps = 1.3$
BIRCH	$T = 2.613, branching = 8, maxLeaf = 8$
D-Stream	$gridsize = 3.5764, Cm = 1.06, Cl = 0.1$
MCOD	$r = 2.215, t = 13$



(a) Evolving scenario

(b)

(c)

Fig. 11 Evolving data stream test case

Intel processor with 16GB of RAM. The aggressive SHC template is the slowest, due to the bigger neighbourhood (19) that needs to be processed.

5.4 Test case 4: Evolving data stream

The final synthetic test case is an evolving data stream, whose scenario is shown in Fig. 11a. Each scenario step involves 300 data instances. The evolving scenario includes one stationary and two drifting clusters. Drifting clusters begin their movement in the same place and move in opposite directions. At some point in the testing data stream, two drifting clusters get separated and become stationary. Eventually, in the last step of the evolving scenario, we introduce five individual data objects on the path of drifting clusters. Clustering algorithms should recognize these five data objects as outliers.

We performed the multi-objective hyper-parameter tuning for CRI and OJI. Used parameters are given in Table 6. CRI and OJI assessments were performed only for the last step of the evolving scenario. Figure 11b shows cluster recognition, while Fig. 11c shows outlier detection by the tested clustering algorithms. SHC and DBSTREAM are showing

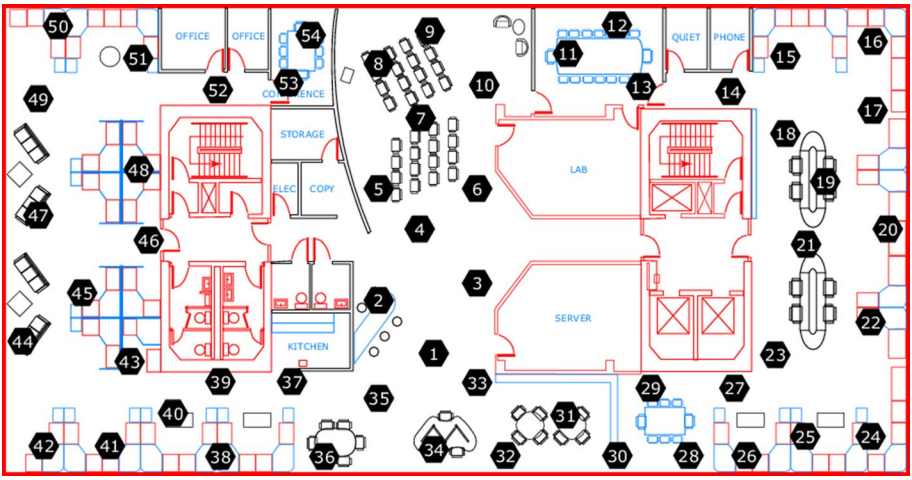


Fig. 12 Sensor placement

Table 7 Algorithm parameters for the sensor data stream

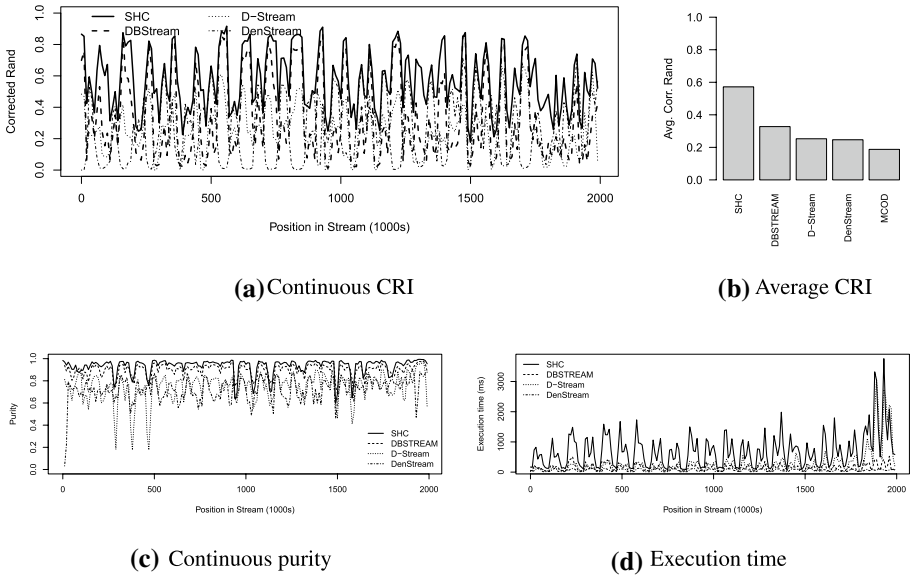
Algorithm	Setting
SHC	$\theta = 4, {\sigma_v}^2 = 0.789, \delta = 31, p_{cb} = 20, p_d = 0.732, \theta_d = 0.258, {\sigma_{\theta_{min}}}^2 = 0.476$
DBSTREAM	$r = 0.743, Cm = 0.41, alpha = 0.0964, shared\_density = true$
D-Stream	$gridsize = 2.45, Cm = 11, Cl = 0.553, attraction = true$
MCOD	$r = 1.1, t = 20$

the best results overall. Due to the low threshold parameter T, BIRCH results are better in the outlier detection than in clustering. MCOD does not perform too well when hyper-parameter tuned both for CRI and OJI, a situation that we already described in the previous test case.

5.5 Test case 5: Intel Berkeley Laboratory sensor data

As one of the real testing data streams, we have used the Intel Berkeley Laboratory sensor data[1]. The placement of the sensors can be seen in Fig. 12. 54 sensors that record temperature, humidity, and light were placed on a floor of the lab. Each sensor also recorded its own accumulator voltage. Sensors were deployed and recording values in over a month period. The main idea is to be able to distinguish and classify distinct sensors. Daily events such as day or night, the sun passing around the building, heating, turning lights on and off, can greatly help to distinguish a part of the building. Many sensors are having missing values due to their power source. This requires some data cleaning. We removed only readings that had missing timestamps. All missing values that can cause errors in some clustering

[1] http://db.csail.mit.edu/labdata/labdata.html



(a) Continuous CRI

(b) Average CRI

(c) Continuous purity

(d) Execution time

Fig. 13 Sensors data stream test case

algorithms, were replaced by zero values. We scaled the data stream the same way as in Hahsler and Bolaños (2016).

We included only algorithms that processed the sensor data stream within the time limit of 2 hours. All algorithms were hyper-parameter tuned for CRI on the first 200000 data objects in the data stream. Parameters used for the test are given in Table 7.

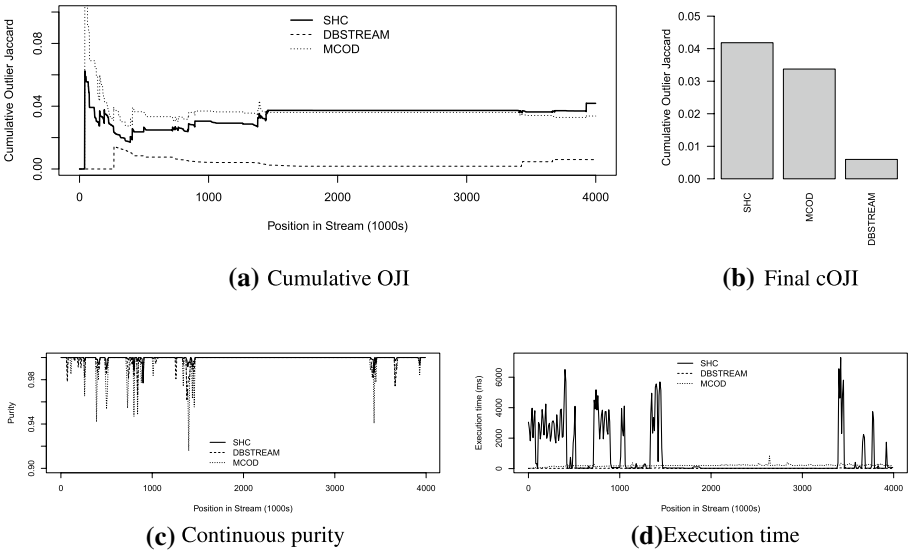
Figure 13a shows continuous CRI for all four major algorithms. SHC is the only continuous algorithm that immediately returns classification for each data object retrieved from the data stream. All other algorithms were processed in batches consisting of 1000 data objects. In Fig. 13b we can see average CRI for the whole sensor data stream. We can see that SHC is the best algorithm when it comes to classification accuracy. Figure 13c shows continuous purity values throughout sensor data stream processing. Continuous execution time was done in 1000 data object steps for all algorithms. Figure 13d shows that SHC is the slowest algorithm to process the sensor data stream, within algorithms that succeeded to process it under the imposed time limit.

5.6 Test case 6: Anomaly detection in KDDCup '99 data stream

KDDCup '99 data stream (Dua and Graff 2017) is one of the most used in testing clustering algorithms. This is a data stream made of network traffic that was used to detect and analyse network intrusions. Most of the KDDCup '99 data stream consists of normal network traffic. Each attack has its own type. An attack can occur is fast bursts, i.e., a set of network communications in a short time frame belongs to the same attack. By modifying the *stream* R package, we can mark outliers in the input data stream, similar to how classes are marked. We marked each attack first occurrence as an outlier, or anomaly that needs to be detected by the clustering algorithms.

Table 8 Algorithm parameters for the KDDCup '99 data stream

Algorithm	Setting
SHC	$\theta = 34, \sigma_v{}^2 = 14.4, \delta = 194, \sigma_{\theta_{min}}{}^2 = 0.144, \theta_{sp} = 19$
DBSTREAM	$r = 1.91, Cm = 3.99, shared\_density = true$
MCOD	$r = 97.8, t = 100$



(a) Cumulative OJI **(b)** Final cOJI

(c) Continuous purity **(d)**Execution time

Fig. 14 KDDCup '99 data stream test case

KDDCup '99 is not the best data stream for testing clustering algorithms since its classes do not correlate tightly with most external clustering indices, such as CRI (Desgraupes 2017), because there are big data stream sections where only one class can be found in data objects, e.g., normal network traffic. On the other side, internal clustering indices do not accurately reflect clustering algorithm accuracy for this data stream.

We use this data stream to demonstrate anomaly detection capabilities of clustering algorithms. For this purpose, we use the cumulative OJI on the KDDCup '99 data stream having marked attack outliers as previously described. We performed hyper-parameter tuning for all involved clustering algorithms for the first 200000 data objects in the KDDCup'99 data stream. Parameters are available in Table 8.

The window size for MCOD plays a crucial role in hyper-parameter tuning, especially since we have 4 million data objects in the data stream. Contrarily to SHC where decay is beneficial, shortening window size in MCOD did not bring better outlier detection accuracy, it only improved speed and memory consumption. The best result for MCOD is obtained for a very wide window, comprising the whole data stream.

Figure 14a shows the cumulative OJI. Since OJI (41) takes into account false positive and undetected outliers, algorithm parameters, especially decay and fading functions, can have great influence over the cOJI results. Shortening decay means a greater generation of false positive detections, while prolonging decay means less true positive outliers. In the

Table 9 Capability to test case mapping

	Static clustering	Stream clustering	Outlier detection	Evolution tracking	Speed
Seeds	✓				
Static clusters and outliers	✓		✓		✓
Evolving stream		✓	✓	✓	
Sensor		✓	✓	✓	✓
KDDCup '99		✓	✓	✓	✓

Table 10 Algorithm capability grades

	Supply k	Static clustering	Stream clustering	Outlier detection	Evolution tracking	Speed
SHC	×	★★★★	★★★★★	★★★★★	★★★★★	★★★
DBSTREAM	×	★★★★	★★★★★	★★	★★★	★★★★★
DBSCAN[a]	×	★★★★★	★★★★	○	★★	★★
MCOD	×	★★	★★	★★★★★	★★	★★★
CluStream[a]	✓	★★	★★	○	★	★★★★★
D-Stream	×	★★★	★★★	★★	★★★	★★★★
DenStream	×	×	★★	×	×	★★★★
BIRCH[a]	×	★★★★	★★★★	★	★★	★★★★★
k-means[a]	✓	★★★★★	★★	○	★	★

[a] Did not process sensor dataset under 2 hours limit

case of MCOD, window size (w) is the only vague parameter that could potentially, correctly tuned, give better results. Figure 14b shows the final cOJI for the whole KDDCup '99 data stream. Figure 14c shows continuous purity throughout data stream processing, characterized by occasional drops in purity when the concentration of outliers and attacks rises. Finally, Fig. 14d shows execution time for all algorithms. MCOD seems to slower than DBTSTREAM. SHC is the slowest in cases when the algorithm starts processing bursts of outliers. Although it looks like cOJI values are quite small, this depends on the processed data stream.

5.7 Evaluation summary

We divided our testing into five distinct major categories, each representing notable capability, or a set of capabilities. The capability categories are:

- **Static clustering** - Capability of clustering finite datasets that do not exhibit population evolution and fading or decay is not needed,
- **Stream clustering** - Clustering of endless data streams that comprise population evolution and fading or decay is needed,
- **Outlier detection** - Capability to accurately detect outliers,
- **Evolution tracking** - Capability of accurately tracking population evolution, including appearance of outliers,

Table 11 SHC major steps to capability mapping

	Static clustering	Stream clustering	Outlier detection	Evolution tracking	Complexity
Decay		✓	✓	✓	$O(mk)$
Classification	✓	✓	✓		$O(mk)$
Agglomeration	✓	✓	✓		$O(mk_2)$
Model update	✓	✓	✓		$O(m)$
Drift and split		✓		✓	$O(mp_n \max_{co_c \in Cm} p(co_c))$
Neighbourhood processing	✓	✓			$O(m\Phi(-1) \max_{co_c \in Cm} p(co_c))$
Removing obsolete	✓	✓	✓	✓	$O(m(2 + (\delta_{max} - 1)p_{cb}))$

- **Speed** - Capability of timely processing data coming from a dataset or data stream.

In Table 9 we can see mapping between clustering algorithm capabilities and performed testing. The testing results, CRI, OJI and execution speed were summarized, and algorithm grades were calculated from these results, which can be seen in Table 10. We used indices absolute range for grading, and for that specific reason there are no algorithms graded with one ★ in the clustering capabilities, as all of them are performing clustering more or less successfully, i.e., none of the tested algorithms did not perform with CRI close to 0. Some of the tested algorithms were graded with ○ in the outlier detection capability, since they did not manage to get OJI > 0. Overall, two best data streaming clustering algorithms are DBSTREAM and SHC, since they perform equally good in all areas, with SHC being slightly better in the outlier detection and population evolution tracking, and DBSTREAM being overall faster than SHC.

Finally, Table 11 gives mapping between SHC major steps and capabilities. In the same table we have the computational complexity for all SHC major steps.

6 Conclusion

In this paper, we proposed a statistical hierarchical clustering algorithm, capable of processing evolving data streams and detecting anomalies. The proposed SHC algorithm comprises all modern clustering algorithm features. It is capable of detecting normally distributed populations and their outliers. Also, drift and split allow SHC to process evolving data streams. The ultimate goal behind creating SHC was to get a general purpose clustering algorithm that performs equally well in clustering and outlier detection, which is experimentally confirmed with several test cases. The classification results match top clustering algorithms in this area.

The big difference is the way SHC works. It is a sequential clustering algorithm that works in a single phase, giving immediate classification on the component, outlier, and cluster levels. This is important from the anomaly detection point of view, as outliers should be further investigated as soon as possible. The single-phase approach also significantly contributes to all sequential calculations, such as building the coincidence matrix.

This might be perceived as an unfair advantage over the two-phase clustering algorithms regarding the accuracy of classification since the offline phase is not supposed to be invoked for every data object retrieved from the processed data stream. We have to look at the trade-off in the speed of the data stream processing. Due to the algebraic simplicity, two-phase clustering algorithms have a fast online phase, which does not give the final classification.

An interesting novel concept developed for SHC is the hierarchical clustering used for detecting and analysing component population evolution. A window of newly retrieved component sub-population is clustered by another child SHC instance, to analyse how the component is evolving and whether it is stationary, drifting, or even splitting into several sub-populations. All the big components from the child SHC instance are then taken and assessed. By calculating the drift index we can detect whether new sub-clustered components are moving away from the original parent component position. If so, we restructure the components and outliers to accommodate population evolution.

Component agglomeration and growth are often accompanied by outlier and component inclusions. To limit the space needed for searching outliers and components that can be included, forming and using the statistical neighbourhood is used to speed up the search. Forming the statistical neighbourhood is linked to data objects classification step by reusing classification calculations. SHC could be further advanced by forming a complete statistical tree, similar to M-tree, that could be used for each initial query. Statistical neighbourhood concept could be used to achieve this goal. Creating a statistical query tree for algorithms such as SHC is a topic for the follow-up research.

Another advantage of SHC is classifier identifier traceability. Once a classifier identifier has been returned, it represents a unique class that does not change. This allows data stream evolution tracking, which might not be supported by all two-phase algorithms, especially those that use another external algorithm in the offline phase. Using an independent clustering algorithm in the offline phase, such as DBSCAN or k-means, does not ensure that assigned macro-clusters can be tracked between two offline phase invocations, especially if an online phase was invoked between them, i.e., a model update happened. Potentially, a streaming clustering algorithm could establish traceability between stateful micro-cluster level and stateless macro-cluster level.

While the complexity of SHC is higher compared to other algorithms mentioned in this paper, its execution can be further optimized. Some of the SHC procedures, such as the statistical distance calculations and sub-clustering, can be massively parallelized. Also, modern data science problems require somewhat more complex solutions than old clustering algorithms, such as *k-mean*. Some of the newer clustering algorithms, such as DBSTREAM, represent a step ahead. Modern hardware certainly offers enough processing power and memory for hosting complex clustering algorithms. This is a prospective research subject, especially from the massive parallel processing point of view.

Appendix A: Main procedure detailed pseudo-code

Algorithm 1 SHC main processing procedure

function PROCESS($X, cOnly$)
  if $cOnly = 0 \wedge \delta > 0$ **then** ▷ If model update is allowed and decay is turned on
    PROCESS_DECAY()
  $V = \{co \in Co : (\sigma_{\theta_{max}}{}^2 = 0 \vee \theta_{lim}(\Sigma(co), \sigma_{\theta_{max}}{}^2) = 0) \wedge Obsolete \notin \phi(co)\}$
    ▷ All classification objects under the $\sigma_{\theta_{max}}{}^2$ variance limit (12) and not obsolete
  $(Q_c, Q_n) = \text{CLASSIFY}(X, V)$ ▷ (19)
  $co_c = \underset{x \in Q_c}{arg\ min}\ d(x)$ ▷ Get the component from Q_c where X is statistically closest to the center (20)
  if $cOnly = 0$ **then** ▷ If model update is allowed
    for $(q_1, q_2) \in (Q_c \times Q_c) : co(q_1) \neq co(q_2)$ **do** ▷ For all distinct pairs in Q_c
      $co_r = \text{AGGLOMERATE}(co(q_1), co(q_2))$ ▷ Agglomerate
      if $co_r \neq \varnothing$ **then**
        $Q_c = Q_c \setminus (co_r, *)$ ▷ If a classified object from Q_c was outlier and removed
        if $co_c = co_r$ **then**
          $co_c = \underset{x \in Q_c}{arg\ min}\ d(x)$ ▷ co_c was removed, recalculate (20)
    if $|Q_c| = 0$ **then** ▷ If the classified set is empty
      $co_c = \text{NEW\_OUTLIER}(X, \sigma_v{}^2)$ ▷ Create a new outlier (11)
    else
      UPDATE(co_c, X, Q_c) ▷ Update model of the closest classifier object co_c
    for $n_i \in Q_n$ **do** ▷ Iterate through the *neighbourhood set*
      $N(co_c) = N(co_c) \cup n_i$ ▷ Add each neighbour to co_c (24)
    PROCESS_NEIGHBOURHOOD(co_c) ▷ Process neighbourhood for co_c
    PROCESS_OBSOLETE() ▷ Remove obsolete class. objects
  return (co_c, Q_c)

Detailed pseudo-code of the SHC main processing procedure is given in Algorithm 1. The main procedure takes two parameters: an input data object X and a *classification only* flag $cOnly \in \{0, 1\}$. The classification only flag indicates whether there is learning or not. For $cOnly = 1$ SHC performs only classification of the input data object X.

If $cOnly$ allows model update and decay factor is set $\delta > 0$ we perform decay check for all classification objects in Co as the first step. After decay has been performed, we select a subset of classification objects $V \subseteq Co$ suitable and eligible for the classification, which have to be classification objects under the variance limit $\sigma_{\theta_{max}}{}^2$ and not marked obsolete by the previous decay procedure. After the classification has been performed, we get back classified and neighbourhood sets Q_c and Q_n. Since the classification procedure returns the statistical distance $d_\sigma(X, \mu(co), \Sigma(co))$ for each classified object $co \in Q_c$, we are able to select the closest classified object co_c according to (20).

For all the following steps we must have $cOnly = 0$, since after the classification we mostly update SHC structures and classification objects. The classified set Q_c is then used to agglomerate all classified objects. If the agglomeration procedure removes one of the classified objects, usually an outlier, we need to remove this object from the classified set Q_c as well, and to recalculate the closest classified object co_c. If the classification procedure did not find any classified objects, X can be considered for an outlier and we need to create a new outlier object for it. The creation of a new outlier must be according to (11). Otherwise, the closest classified object co_c model can be updated. The model update procedure contains drifting and splitting steps as well. After updating the model for co_c,

the main processing procedure adds neighbourhood from Q_n to the same co_c and initiates the processing of the neighbourhood. In the last step, all obsolete classification objects are removed.

The SHC main procedure returns the closest classified object co_c as the immediate result of the classification. The model update does not change the previously chosen closest classified object co_c, so the classification is considered to be performed prior to the model update.

Appendix B: Decay procedure detailed pseudo-code

Algorithm 2 Decay procedure

```
1: procedure PROCESS_DECAY
2:     for co ∈ Co do
3:         δ(co) = δ(co) − 1                          ▷ Decrease decay counter
4:         if δ(co) ≤ 0 then
5:             φ(co) = φ(co) ∪ {Obsolete}             ▷ If decayed, mark it obsolete
```

The decay procedure in Algorithm 2 is a simple iteration through all classification objects in Co to decrease decay counters. Decay counters are initially set when a new outlier is created or reset when the closest classified object co_c model update is performed. When the decay counter for the processed classification object expires, we mark the classification object as obsolete.

Appendix C: Classification function detailed pseudo-code

Algorithm 3 Classification function

```
1: function CLASSIFY(X, V)
2:     Q_c = {}, Q_n = {}
3:     for co ∈ V do
4:         d = d_σ(X, μ(co), Σ(co))                   ▷ Calculate the stat. distance (5)
5:         if d ≤ θ then                              ▷ (19)
6:             Q_c = Q_c ∪ (co, d)                    ▷ Component is in the classification set
7:         else if θ < d ≤ p_n θ then                  ▷ (19)
8:             Q_n = Q_n ∪ co                         ▷ Component is in the stat. neighbourhood
9:     return (Q_c, Q_n)
```

The classification procedure in Algorithm 3 is described in Sect. 4.2.1. The goal of the classification procedure is to produce classified and neighbourhood sets Q_c, Q_n for the input data object X, using the set of eligible classification objects V. The set of eligible classification objects V is prepared in the main SHC processing procedure. We iterate through the set of eligible classification objects $co \in V$ and test the statistical distance X has to each eligible classification object co. Based on the calculated statistical distance we place the classification object either in the classified set Q_c, in the neighbourhood set Q_n or nowhere.

The classified set Q_c comprises pairs of classified objects and calculated statistical distance d, which can be later used to recalculate the closest classified object co_c again.

Appendix D: Agglomeration function detailed pseudo-code

Algorithm 4 Agglomeration function

```
1:  function AGGLOMERATE(co_1, co_2)
2:      if co_1 ∈ O then                                   ▷ If co_1 is an outlier
3:          UPDATE(co_2, μ(co_1), {})                      ▷ Add the outlier to the other object
4:          G_t.ADD(co_1 → co_2)                           ▷ Create a trace
5:          φ(co_1) = φ(co_1) ∪ {Obsolete}                 ▷ Mark the outlier obsolete
6:          return co_1                                    ▷ Notify invoker that outlier has been removed
7:      else if co_2 ∈ O then                              ▷ If co_2 is an outlier
8:          UPDATE(co_1, μ(co_2), {})
9:          G_t.ADD(co_2 → co_1)
10:         φ(co_2) = φ(co_2) ∪ {Obsolete}
11:         return co_2
12:     else                                               ▷ If we have two components
13:         cc = G_a.CONNECTION(co_1, co_2)                ▷ Test or create a connection in G_a (16)
14:         sp(cc) = sp(cc) + 1                            ▷ Increment the shared pop. members (16)
15:         if sp(cc) ≥ θ_sp then   ▷ If the shared pop. members threshold reached, as in Section 4.2.2
16:             if p(co_1) > p(co_2) then   ▷ Cluster comps. under the cluster of the bigger one (23)
17:                 G_c.CLUSTER(co_1, co_2)
18:             else
19:                 G_c.CLUSTER(co_2, co_1)
20:     return ∅
```

The agglomeration procedure in Algorithm 4 is described in Sect. 4.2.2. After we detected the classified set Q_c, and there are multiple classified objects where X could be classified into, i.e., $|Q_c| > 1$, this potentially means that X can be classified to a single cluster $C_i \in Cl$, which can be deducted from the container tree G_c. The agglomeration procedure must ensure the correct cluster structure taking into account the shared population threshold θ_{sp}. The input parameters for the agglomeration procedure are two distinct classified objects, whose distinctiveness is ensured in the main SHC processing procedure.

If at least one of the input classified objects is an outlier, we perform the outlier inclusion, as described in Sect. 4.2.3, into the other classified object without ensuring θ_{sp} threshold. Outliers usually have low virtual covariance (10), and in the case of a shared population member appearance, we can be highly confident that the outlier is a member of a bigger population. After the outlier inclusion by updating the model od the other classified object, we mark the outlier as obsolete and return back the removed outlier to know that the recalculation of the closest classified object co_c is needed.

In case we need to agglomerate two components, we need to work on the agglomeration graph G_a. We test for a connection between components by invoking the agglomeration graph CONNECTION method. If there is no connection between components co_1 and co_2, the CONNECTION method creates a new connection $E_a = E_a \cup \{cc = (co_1, co_2, sp = 0)\}$ in the agglomeration graph. The shared member population counter $sp(cc)$ is initially set to 0, as it will be immediately incremented by the agglomeration procedure. If the shared member population $sp(cc)$ is above the threshold θ_{sp}, we need to restructure the container

tree G_c, so that components co_1 and co_2 belong to the same cluster. This is done by invoking the container tree CLUSTER method. The CLUSTER method must also clean up all vertical container tree paths that are not according to (14), i.e., do not have a classification object for the leaf.

Appendix E: Model update detailed pseudo-code

Model update in Algorithm 5 is described in Sects. 3 and 4.3. The model updating procedure is guarded by the ${\sigma_{\theta_{min}}}^2$ and ${\sigma_{\theta_{max}}}^2$ variance limits. ${\sigma_{\theta_{min}}}^2$ is used to limit outlier to component promotion, to ensure that a newly formed component has some minimal variance. ${\sigma_{\theta_{max}}}^2$ is optional and used only when we want to limit component growth, to achieve .

The classification object co distribution model update is done by updating $\mu(co)$ and $\Sigma(co)$ according to (8) and incrementing the population $p(co)$. At the same time we reset the decay counter $\delta(co)$.

Drift and split

Drift and split mechanism is part of the model updating procedure in Algorithm 5, described in Sect. 4.3. First, we need to create the component co baseline. If the baseline is not created, we wait for the component to grow to satisfy (28). Before starting sub-clustering in the child SHC, i.e., drift and split evaluation, we use a counter to wait for another $p(cb(co))$ data objects that update the component. We process data objects in the child SHC, to achieve component sub-clustering, which is then used to detect significant population evolution. We maintain the connections between classified components in Q_c and $Q_c(SHC(co))$ in the agglomeration graph. Once the biggest component in the child SHC reaches threshold (31), we calculate the drift index (30) and take the final drift and split decision based on (32,34).

If drifting is detected, we move all components from the child SHC into normal components, as in Figs. 5b, c. After this, we restructure the container tree G_c using the connections made in the agglomeration graph G_a. Eventually, we mark the original component co as obsolete and let the sub-clustered components reflect the population(s) evolution.

If drifting is not detected, we remove the child SHC and all connections to the sub-clustered components in the agglomeration graph G_a. We re-instantiate the child SHC for future use. The waiting counter is reset and we wait for $p(\text{Update procedure}cb(co))$ new data objects until new drift and split evaluation.

In Algorithm 5 we introduced some additional methods for the agglomeration graph G_a. ADJACENT_NODES returns a set of adjacent connected components in G_a for the supplied component. REMOVE method is used to remove a set of nodes and their adjacent edges from the agglomeration graph G_a.

Algorithm 5 Update procedure

procedure UPDATE(X, co, Q_c)
  if ${\sigma_{\theta_{min}}}^2 = 0 \vee \theta_{lim}(\Sigma(co), {\sigma_{\theta_{min}}}^2) = 1$ **then** ▷ Check the variance limit (12)
    if $R(co) \neq \varnothing$ **then**
      UPDATE($X, R(co), Q_c$) ▷ Redirect the update (26)
    else
      $\mathcal{N}(\mu(co), \Sigma(co))$.UPDATE($X$) ▷ Update co distribution (8) with data object X
      $p(co) = p(co) + 1$ ▷ Increment the population
      $\delta(co) = \delta \min(p_{cb}, p(co))$ ▷ Reset the decay counter (35)
      if $cb(co) = \varnothing \wedge ((0 < p(co) \leq p_{cb}) \vee ({\sigma_{cb}}^2 > 0 \wedge \theta_{lim}(\Sigma(co), {\sigma_{cb}}^2) = 1)$ **then**
        ▷ (28)
        $cb(co) \leftarrow co$ ▷ Clone the classifier object current snapshot
        $c_1 = p(cb(co))$ ▷ Reset the counter for sub-clustering
      else if $cb(co) \neq \varnothing$ **then**
        if $c_1 > 0$ **then**
          $c_1 = c_1 - 1$ ▷ Count the co pop. members until sub-clustering
        else
          $R = SHC(co)$.PROCESS($X, 0$) ▷ Sub-cluster the component pop. member
          $Q_{c_p} = Q_c \setminus (O \cup \{co\}), Q_{c_{sub}} = Q_c(R) \setminus O(SHC(co))$ ▷ Remove outliers (16,33)
          for $(q_p, q_c) \in (Q_{c_p} \times Q_{c_{sub}})$ **do** ▷ All component pairs in the *classified sets*
            $cc = G_a$.CONNECTION($co(q_p), co(q_c)$)
            ▷ Test or create connection between this component and sub-clustered component (33)
            $sp(cc) = sp(cc) + 1$ ▷ Increment the shared population
          $cm_{biggest} = \underset{cm_i \in Cm(SHC(co))}{arg\,max} \; p(cm_i)$ ▷ (29)
          if $p(cm_{biggest}) > p_d p(cb(co))$ **then** ▷ (31)
            $d_i = \sum_{cm_i \in Cm(SHC(co))} p(cm_i) d_\sigma(\mu(cm_i), \mu(cb(co)), \Sigma(cb(co)))$ ▷ (30)
            if $d_i \geq \theta_d \theta$ **then** ▷ Drifting detected (32)
              for $cm_t \in Cm(SHC(co))$ **do** ▷ Set of activities seen in Figures 5b and 5c
                $Cm = Cm \cup \{cm_t\}$
                G_t.ADD($co \rightarrow cm_t$)
                $Cm' = G_a$.ADJACENT_NODES(cm_t)
                for $cm_s \in Cm'$ **do**
                  $cc = G_a$.CONNECTION(cm_s, cm_t)
                  if $sp(cc) \geq \theta_{sp}$ **then**
                    G_c.CLUSTER(cm_s, cm_t)
              $\phi(co) = \phi(co) \cup \{Obsolete\}$ ▷ Mark the current object obsolete
            else ▷ Drifting not detected (34)
              G_a.REMOVE($Cm(SHC(co))$)
              ▷ Clean up the G_a by removing sub-clustered components
              $SHC(co) =$ new SHC ▷ Re-instantiate the child SHC
              $c_1 = p(cb(co))$ ▷ Reset the counter for sub-clustering

Appendix F: Neighbourhood processing detailed pseudo-code

Algorithm 6 Neighbourhood processing procedure

procedure PROCESS_NEIGHBOURHOOD(co)
  if $co \in Cm$ **then** ▷ We process only components
    if $\theta_{lim}(\Sigma(co), {\sigma_{\theta_{max}}}^2) = 0$ **then**
      for $n \in N(co)$ **do** ▷ Iterate through all neighbours
        if $R(n) = co \wedge p(n) < p_{rr} p(co)$ **then**
          G_t.ADD($n \rightarrow co$) ▷ Add trace
          $\phi(n) = \phi(n) \cup \{Obsolete\}$
          ▷ Remove redirected neighbour as defined in Section 4.2.3
        else if $d_\sigma(\mu(n), \mu(co), \Sigma(co)) \leq \theta$ **then**
          if $p(n) = 1$ **then**
            AGGLOMERATE(co, n) ▷ Outlier inclusion (25)
          else
            $R(n) = co$ ▷ Component redirection (26)
    else
      for $n \in N(co)$ **do** ▷ Iterate through all neighbours
        if $R(n) = co$ **then**
          $R(n) = \varnothing$ ▷ Remove redirections
      $N(co) = \emptyset$ ▷ Clean the neighbourhood

Neighbourhood processing in Algorithm 6 is described in Sect. 4.2.3. The neighbourhood set Q_n produced in the classification procedure, Algorithm 3, is constantly updating the neighbourhood of components in Cm. We process entire neighbourhood of the closest classified component $co \in Cm$ that did not reach the variance limit ${\sigma_{\theta_{max}}}^2$. If the variance limit ${\sigma_{\theta_{max}}}^2$ has been reached for co, we clean all redirections to co and neighbourhood $N(co)$, since the neighbourhood processing for these components is not performed.

If we are still below the variance limit ${\sigma_{\theta_{max}}}^2$, or component growth is not limited by ${\sigma_{\theta_{max}}}^2 = 0$, we iterate through the whole neighbourhood $n \in N(co)$. If the neighbour n is redirected to component co and its population falls behind the population of co (27), we mark this neighbour as obsolete.

In case if neighbour n is an outlier we perform the outlier inclusion, adding the outlier n to the component co population. If neighbour n is a component whose center $\mu(n)$ is statistically closer than θ we do the component inclusion by redirecting the component n to the component co.

Appendix G: Removing obsolete classification objects detailed pseudo-code

Algorithm 7 Removing the obsolete classification objects

```
1:  procedure PROCESS_OBSOLETE
2:      for co_o ∈ Co : Obsolete ∈ φ(co_o) do
3:          N_a = G_a.PARTITION()                          ▷ Partition the agglomeration graph G_a (36)
4:          if ∃N_{a_i} ∈ N_a : co_o ∈ N_{a_i} then        ▷ Isolate the sub-graph comprising co_o
5:              G_a' = G_a[N_{a_i} \ {co_o}]               ▷ Vertex-induced subgraph (37)
6:              N_{a_i}' = G_a'.PARTITION()                ▷ Partition the agglomeration graph without co_o (38)
7:              for (n_1, n_2) ∈ (N_{a_i}' × N_{a_i}') : n_1 ≠ n_2 do
8:                  G_c.DECLUSTER(n_1, n_2)                ▷ Separate partitions in the container tree G_c
9:          G_c.REMOVE(co_o)                               ▷ Remove co_o from the container tree G_c
10:         G_a.REMOVE(co_o)                               ▷ Remove co_o and adjacent edges from G_a
11:         for co_i ∈ Co : Obsolete ∉ φ(co_i) do          ▷ Clean up redirection and neighbourhood lists
12:             if co_o = R(co_i) then
13:                 R(co_i) = ∅
14:             if co_o ∈ N(co_i) then
15:                 N(co_i) = N(co_i) \ {co_o}
```

Removing obsolete classification objects is detailed in Algorithm 7 and described in Sect. 4.5. We iterate through all obsolete classification objects. First, we partition G_a nodes and find the partition N_{a_i} having the obsolete object co_o we are about to remove. Then we remove co_o from the partition N_{a_i} and create a vertex-induced subgraph $G_a{}'$ from it. We partition $G_a{}'$, and if there are more partitions in it, we de-cluster G_c according to partitions found in $G_a{}'$. De-clustering is done by restructuring G_c so that partitions of $G_a{}'$ reside in separate clusters within G_c. For this purpose, we need the container tree G_c DECLUSTER method, which takes two partitions as input parameters. We separate these partitions in the container tree, leaving the more populated one in the old cluster node. Eventually, after separating all distinct partition pairs, the most populated partition must remain in the original, starting cluster node.

After this, we remove the obsolete classification object co_o from both the container tree and agglomeration graph. We introduce the container tree G_c REMOVE method that helps us removing classification objects and adjacent edges from the container tree. The REMOVE method must also clean up all vertical container tree paths that are not according to (14), i.e., do not have a classification object for the leaf.

In the end, we iterate through all remaining classification objects, i.e., those that are not obsolete, and remove the obsolete classification object co_o from all redirections and neighbourhoods.

Appendix H: The maximum number of components and outliers

All current classification objects $k = |Co| = k_c + k_o$ can be divided into components $k_c = |Cm|$ and outliers $k = |O|$. Estimating the maximal number of classification objects is a combinatorial problem of analysing the worst case scenario for components and outliers in the

processed data stream. The maximal number of outliers can be achieved only when there is a window w_1 of the latest data objects, such that $w_1 = k_o$ all of them are outliers, and the oldest outlier decay counter is set to $\delta(o) = 1$ at the end of the window w_1. According to (35) this window can comprise $w_1 = \delta_o \delta p_{cb}$ data objects.

$$k_o = \delta_o \delta p_{cb} \tag{42}$$

For all components that were formed before this window, we know that they needed to have population big enough not to decay in the outlier forming window w_1, i.e., $\min_{cm \in Cm} \delta(cm) = w_1 + 1$. Before window w_1 we need to have a window w_2 where we reset all decay counters for formed components. The oldest component, i.e., the one appearing at the beginning of the window w_2, must not decay through both windows $w_2 + w_1$. The question is how many components we can create before reaching windows $w_2 + w_1$? According to (35), we have set boundaries for component decay to $[\delta p_{cb}, \delta_{max} \delta p_{cb}]$. This means that we will eventually hit the upper limit of

$$k_c = \delta_{max} \delta p_{cb} \tag{43}$$

components, since there is no combination of the input data objects in the processed stream that would allow us to have more components. In case $\delta_{max} > \delta_o$ the combination of windows $w_2 + w_1$ cannot be bigger than k_c, otherwise the oldest component would decay. This means that we can have either $w_2 = k_c \wedge w_1 = 0$ or $w_2 = k_c - k_o \wedge w_1 = k_o$. Otherwise, if we have $\delta_{max} \leq \delta_o$, the only possibility is to have only outliers in the combination $w_2 = 0 \wedge w_1 = k_o$, as none of the previously formed components would not decay in the window w_1. Finally, the maximal number of classification objects is

$$k = \max(\delta_o, \delta_{max}) \delta p_{cb} \tag{44}$$

However, to meet the worst case scenario in the neighbourhood processing, we prefer the window w_1 to be fully populated with outliers.

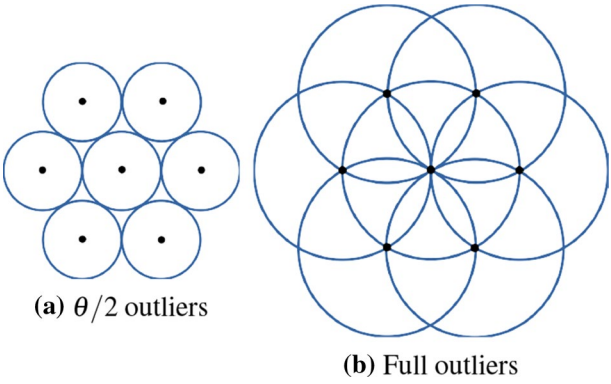


Fig. 15 Two-dimensional outlier kissing number example

Appendix I: Outliers packing and kissing number

When we estimate the maximal number of classified objects in the classified set Q_c, we start from the most dense packed set. This is most definitely a set of outliers, where each outlier is being characterized by a θ bound σ_v hypersphere.

To solve this, we consult the kissing number $\mathcal{K}(d)$ (the Newton's number) of an outlier hypersphere reduced to $\theta/2$, where d is the number of space dimensions. Such kissing number gives the possible number of outliers that can be packed around the outlier hypersphere. Such example can be seen in Fig. 15a. By fully expanding outliers back to θ, the central outlier can be perceived as a data object that can be classified to all kissing outliers. This can be seen in Fig. 15b. This also means that the maximal classified set is the set of outliers packed this way, having $|Q_c| = \mathcal{K}(d)$.

If we replace only one outlier in the Fig. 15 with component having $\Sigma > \Sigma_v$, then the number of kissing classification objects will be less than $\mathcal{K}(d)$, as we expect the component hypersphere radius to be bigger than the outlier hypersphere.

References

Ackerman, M., & Dasgupta, S. (2014). Incremental clustering: The case for extra clusters. In *Advances in Neural Information Processing Systems* (pp. 307–315).

Aggarwal, C. C. (2007). *Data streams: Models and algorithms* (Vol. 31). New York: Springer.

Aggarwal, C. C., Han, J., Wang, J., & Yu, P. S. (2003). A framework for clustering evolving data streams. In *Proceedings of the 29th international conference on Very large data bases-Volume 29, VLDB Endowment* (pp. 81–92).

Baudry, J. P., Raftery, A. E., Celeux, G., Lo, K., & Gottardo, R. (2010). Combining mixture components for clustering. *Journal of Computational and Graphical Statistics*, *19*(2), 332–353.

Bifet, A., & Gavalda, R. (2007). Learning from time-changing data with adaptive windowing. In *Proceedings of the 2007 SIAM international conference on data mining, SIAM* (pp. 443–448).

Bifet, A., Read, J., Holmes, G., & Pfahringer, B. (2018). Streaming data mining with massive online analytics (MOA). *Series in Machine Perception and Artificial Intelligence*, *83*(1), 1–25.

Bischl, B., Richter, J., Bossek, J., Horn, D., Thomas, J., & Lang, M. (2017). mlrMBO: A modular framework for model-based optimization of expensive black-box functions. 1703.03373

Bradley, P. S., Fayyad, U. M., Reina, C., et al. (1998). Scaling clustering algorithms to large databases. In *KDD-98* (pp. 9–15).

Cao, F., Estert, M., Qian, W., & Zhou, A. (2006). Density-based clustering over an evolving data stream with noise. In *Proceedings of the 2006 SIAM international conference on data mining, SIAM* (pp. 328–339).

Cao, L., Yang, D., Wang, Q., Yu, Y., Wang, J., & Rundensteiner, E. A. (2014). Scalable distance-based outlier detection over high-volume data streams. In *2014 IEEE 30th international conference on data engineering (ICDE), IEEE* (pp. 76–87).

Carnein, M., Assenmacher, D., & Trautmann, H. (2017). An empirical comparison of stream clustering algorithms. In *Proceedings of the computing frontiers conference, ACM* (pp. 361–366).

Chen, Y., & Tu, L. (2007). Density-based clustering for real-time stream data. In *Proceedings of the 13th ACM SIGKDD international conference on knowledge discovery and data mining, ACM* (pp. 133–142).

Ciaccia, P., Patella, M., & Zezula, P. (1997). M-tree: An efficient access method for similarity search in metric spaces. In *Proceedings of the 23rd VLDB conference*, Athens, Greece, Citeseer (pp. 426–435).

Conway, J. H., & Sloane, N. J. A. (2013). *Sphere packings, lattices and groups* (Vol. 290). New York: Springer.

Dasgupta, S. (1999). Learning mixtures of gaussians. In *40th annual symposium on Foundations of computer science, 1999, IEEE* (pp. 634–644).

De Maesschalck, R., Jouan-Rimbaud, D., & Massart, D. L. (2000). The mahalanobis distance. *Chemometrics and Intelligent Laboratory Systems*, *50*(1), 1–18.

Desgraupes, B. (2017). Clustering indices. *University of Paris Ouest-Lab Modal'X*, *1*, 34.

Dua, D., & Graff, C. (2017). UCI machine learning repository. http://archive.ics.uci.edu/ml.

Eddelbuettel, D., & Balamuta, J. J. (2018). Extending R with C++: A brief introduction to Rcpp. *The American Statistician*, *72*(1), 28–36. https://doi.org/10.1080/00031305.2017.1375990.

Ester, M., Kriegel, H. P., Sander, J., Xu, X., et al. (1996). A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD-96, AAAI* (pp. 226–231).
Fichtenberger, H., Gillé, M., Schmidt, M., Schwiegelshohn, C., & Sohler, C. (2013). BICO: BIRCH meets coresets for k-means clustering. In *European symposium on Algorithms* (pp. 481–492). New York: Springer.
Gama, J. (2010). *Knowledge discovery from data streams*. Boca Raton: CRC Press.
Gama, J., & Gaber, M. M. (2007). *Learning from data streams: Processing techniques in sensor networks*. New York: Springer.
Gut, A. (2009). *An intermediate course in probability* (2nd ed.). New York: Springer. https://doi.org/10.1007/978-1-4419-0162-0.
Hahsler, M., & Bolaños, M. (2016). Clustering data streams based on shared density between micro-clusters. *IEEE Transactions on Knowledge and Data Engineering*, *28*(6), 1449–1461.
Hahsler, M., Bolanos, M., Forrest, J., et al. (2017). Introduction to stream: An extensible framework for data stream clustering research with r. *Journal of Statistical Software*, *76*(14), 1–50.
Hahsler, M., Bolanos, M., & Forrest, J. (2018). streamMOA: Interface for MOA Stream Clustering Algorithms. https://CRAN.R-project.org/package=streamMOA, R package version 1.1-4.
Horn, D., Wagner, T., Biermann, D., Weihs, C., & Bischl, B. (2015). Model-based multi-objective optimization: Taxonomy, multi-point proposal, toolbox and benchmark. In A. Gaspar-Cunha, C. Henggeler Antunes, & C. C. Coello (Eds.), *Evolutionary multi-criterion optimization* (pp. 64–78). Cham: Springer.
Jaccard, P. (1912). The distribution of the flora in the alpine zone.1. *New Phytologist*, *11*(2), 37–50. https://doi.org/10.1111/j.1469-8137.1912.tb05611.x.
Jacob, B., Guennebaud, G., et al. (2013). Eigen: C++ template library for linear algebra.
Jain, A. K., Murty, M. N., & Flynn, P. J. (1999). Data clustering: A review. *ACM Computing Surveys (CSUR)*, *31*(3), 264–323.
Karypis, G., Han, E. H., & Kumar, V. (1999). Chameleon: Hierarchical clustering using dynamic modeling. *Computer*, *32*(8), 68–75.
Kontaki, M., Gounaris, A., Papadopoulos, A. N., Tsichlas, K., & Manolopoulos, Y. (2016). Efficient and flexible algorithms for monitoring distance-based outliers over data streams. *Information Systems*, *55*, 37–53.
Kranen, P., Assent, I., Baldauf, C., & Seidl, T. (2011). The clustree: Indexing micro-clusters for anytime stream mining. *Knowledge and Information Systems*, *29*(2), 249–272.
Krishnamoorthy, A., & Menon, D. (2013). Matrix inversion using cholesky decomposition. In *2013 signal processing: Algorithms, architectures, arrangements, and applications (SPA), IEEE* (pp. 70–72).
Micceri, T. (1989). The unicorn, the normal curve, and other improbable creatures. *Psychological Bulletin*, *105*(1), 156.
Moshtaghi, M., Leckie, C., & Bezdek, J. C. (2016). Online clustering of multivariate time-series. In *Proceedings of the 2016 SIAM international conference on data mining, SIAM* (pp. 360–368).
Nguyen, H. L., Woon, Y. K., & Ng, W. K. (2015). A survey on data stream clustering and classification. *Knowledge and Information Systems*, *45*(3), 535–569. https://doi.org/10.1007/s10115-014-0808-1.
O'Boyle, E, Jr., & Aguinis, H. (2012). The best and the rest: Revisiting the norm of normality of individual performance. *Personnel Psychology*, *65*(1), 79–119.
Press, W. H., Teukolsky, S. A., Vetterling, W. T., & Flannery, B. P. (2007). *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge: Cambridge University Press.
Rasmussen, E. M. (1992). Clustering algorithms. *Information Retrieval: Data Structures & Algorithms*, *419*, 442.
Sherman, J., & Morrison, W. J. (1950). Adjustment of an inverse matrix corresponding to a change in one element of a given matrix. *The Annals of Mathematical Statistics*, *21*(1), 124–127.
Silva, J. A., Faria, E. R., Barros, R. C., Hruschka, E. R., de Carvalho, A. C., & Gama, J. (2013). Data stream clustering: A survey. *ACM Computing Surveys (CSUR)*, *46*(1), 13.
Solaimani, M., Iftekhar, M., Khan, L., & Thuraisingham, B. (2014). Statistical technique for online anomaly detection using spark over heterogeneous data from multi-source vmware performance data. In *2014 IEEE international conference on Big Data (Big Data), IEEE* (pp. 1086–1094).
Welford, B. P. (1962). Note on a method for calculating corrected sums of squares and products. *Technometrics*, *4*(3), 419–420.
Woodbury, M. A. (1950). Inverting modified matrices. *Memorandum Report*, *42*(106), 336.
Zhang, T., Ramakrishnan, R., & Livny, M. (1996). BIRCH: an efficient data clustering method for very large databases. In *ACM Sigmod Record, ACM* (pp. 103–114).